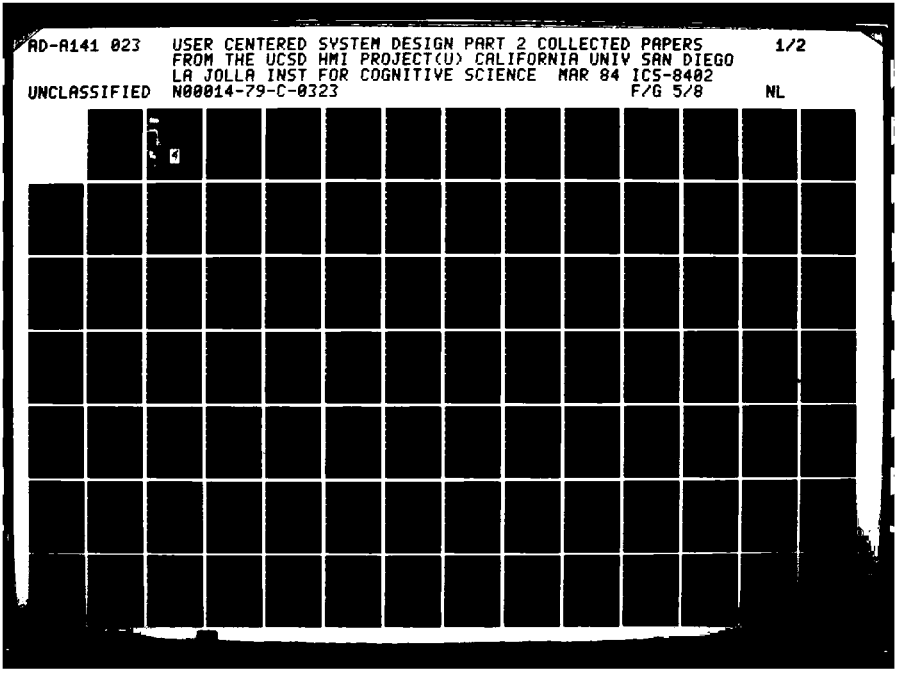
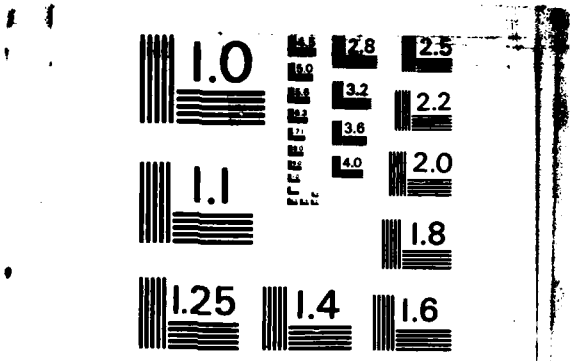
AD-A141 023 USER CENTERED SYSTEM DESIGN PART 2 COLLECTED PAPERS 1/2
FROM THE UCSD HMI PROJECT(U) CALIFORNIA UNIV SAN DIEGO
LA JOLLA INST FOR COGNITIVE SCIENCE MAR 84 ICS-8402
UNCLASSIFIED N00014-79-C-0323 F/G 5/8 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

USER CENTERED SYSTEM [illegible]

COLLECTED PAPERS
from the
UCSD HMI PROJECT

March 1984

ICS Report No. 8402

INSTITUTE FOR COGNITIVE SCIENCE
UNIVERSITY OF CALIFORNIA, SAN DIEGO LA JOLLA, CALIFORNIA 92093

USER CENTERED SYSTEM DESIGN

Part II

COLLECTED PAPERS
from the
UCSD HMI PROJECT

March 1984

ICS Report No. 8402

The papers included in this collection were prepared for several different conferences, including the 7th Annual Conference on Software Engineering, Florida, March 1984, and the First IFIP Conference on Human-Computer Interaction to be held in London in September 1984.

The research reported here was conducted under Contract N00014-79-C-0323, NR 667-437 with the Personnel and Training Research Programs of the Office of Naval Research, and was sponsored by the Office of Naval Research and a grant from the System Development Foundation. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the sponsoring agencies. Approved for public release; distribution unlimited. Reproduction in whole or in part is permitted for any purpose of the United States Government.

ONR REPORT 8402

Requests for reprints should be sent to Donald A. Norman, Institute for Cognitive Science C-015; University of California, San Diego; La Jolla, California, 92093, USA.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ONR #8402	2. GOVT ACCESSION NO. AD-A142 023	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) User Centered System Design: Part II, Collected Papers from the UCSD HMI Project		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER ONR 8402 ICS #8402
7. AUTHOR(s) The UCSD HMI Project		8. CONTRACT OR GRANT NUMBER(s) N00014-79-C-0323
9. PERFORMING ORGANIZATION NAME AND ADDRESS Center for Human Information Processing Institute for Cognitive Science University of California, San Diego La Jolla, CA 92093		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 667-437
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel & Training (Code 442 PT) Office of Naval Research 800 N. Quincy St., Arlington, VA 22217		12. REPORT DATE March 1984 13. NUMBER OF PAGES 120
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This research was also supported by a grant from the System Development Fndn.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

help systems	human-computer interaction
computer system evaluation	information retrieval
documentation	interface design
expert systems	software engineering
expert-novice users	structured activities

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

OVER

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102 LF 014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract

This report is a collection of recent papers by the Human-Machine Interaction group at the University of California, San Diego.

Stages and Levels in Human-Machine Interaction by D.A. Norman
The interaction between a person and a computer system involves four different stages of activities—*intention, selection, execution,* and *evaluation*—each of which may occur at different levels of specification. Analysis of these stages and levels provides a useful way of looking at the issues of human-computer interaction.

The Nature of Expertise in UNIX by S. Draper
This paper discusses the nature of expertise in UNIX, arguing that in certain senses of the word there are no experts. The consequences for interface design of revising the common-sense notion of expertise, particularly with respect to designing help facilities, are then discussed.

Users in the Real World by D. Owen
Based on the premise that people demonstrate a considerable degree of competence at formulating and achieving goals in the world, this paper seeks to identify and examine the relationship between the crucial characteristics of the real world and inherent or acquired human skills that support this competence, in order to improve the human computer interface. Aspects examined include a "naive physics" of computing and the reconstruction of propositionally held information.

Constructive Interaction: A Method for Studying User-Computer-User Interaction by C. O'Malley, S. Draper, and M.S. Riley
This paper describes a promising technique for studying human-machine interaction called Constructive Interaction (Miyake, 1982).It consists essentially of recording sessions with two participants who are discussing some topic which they do not fully understand. Miyake was interested in what was revealed about the underlying schemas of the participants and how new schemas can originate in an interaction between two people. We are interested in what this basic situation can offer for the study of HMI.

Formalizing Task Descriptions for Command Specification and Documentation by P. Smolensky, M.L. Monty, and E. Conway
The problem of formally describing computer tasks in terms of the input given and the output desired is considered. A feasibility study in the domain of printing suggests that *task attributes* provide a powerful language for such descriptions. It is argued that task description is important for moving the center of human-machine interface design away from the machine and toward the user.

Problems in Evaluation of Human-Computer Interfaces: A Case Study by L.J. Bannon and C. O'Malley.
One of the most difficult aspects of interface design is evaluating new or changed features of an interface. This paper evaluates methods of evaluation and design in the context of a program developed to assist users in getting quick access to information contained in the UNIX[1] manual.

Planning Nets: A Framework for Analyzing User-Computer Interactions by M.S. Riley and C. O'Malley
During the course of interacting with a computer, a user has goals that correspond to tasks to be performed and must plan how to achieve those goals with the available commands. A framework for analyzing user goals, the mapping between those goals and available commands, and the factors influencing the success and efficiency of the resulting plans is presented. The implications of this analysis for the development of principles for improving user-computer interactions are discussed.

Activity Scripts by A. Cypher
A session with the computer can be organized around the *activities* of the user, rather than around the *actions* of the computer. A user-centered approach to grouping stereotypical sequences of commands into scripts or macros is discussed. This approach illustrates several issues in Human/Computer Interaction: *joint problem solving, tool/task mismatches,* and *visible effects.*

DESCRIBE: Environments for Specifying Commands and Retrieving Information by Elaboration by S. Greenspan and P. Smolensky
In communication between people, objects and events are principally referred to through *description.* This note argues that the basic principles that make such reference by description possible can also be employed in communication between people and computers. A new type of operating system called *DESCRIBE* in which commands and files are referenced by description (as well as by name) is proposed.

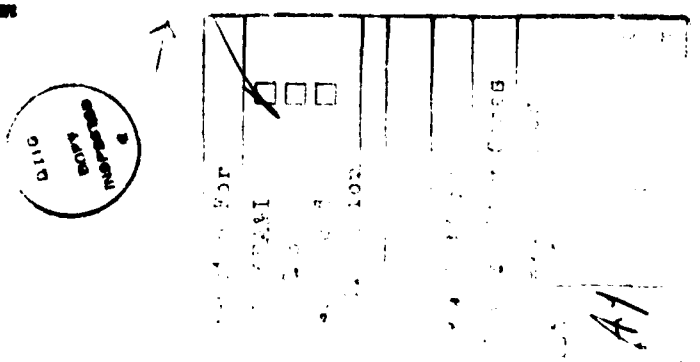
Caveats on the Use of Expert Systems by L.J. Bannon
Recently we have witnessed a round of assertions and counter-assertions about the capabilities of applied AI, specifically in the area called "knowledge engineering," where scientists are involved in the building of so-called "expert systems" that are designed to mimic the performance of human experts in certain domains. Strong claims about the potential social benefits of such systems are being voiced, but this paper is concerned with the caveats.

Software Engineering for User Interfaces by S. Draper and D.A. Norman
The discipline of Software Engineering can be extended in a natural way to deal with the issues raised by a systematic approach to the design of human-machine interfaces. Two main points are made: that the user should be treated as part of the system being designed, and that projects should be organized to take account of the current (small) state of a priori knowledge about how to design interfaces.

USER CENTERED SYSTEM DESIGN:

PART II

COLLECTED PAPERS from the UCSD HMI PROJECT



ABSTRACTS

Stages and Levels in Human-Machine Interaction by *D.A. Norman*

The interaction between a person and a computer system involves four different stages of activities—*intention*, *selection*, *execution*, and *evaluation*—each of which may occur at different levels of specification. Analysis of these stages and levels provides a useful way of looking at the issues of human-computer interaction.

The Nature of Expertise in UNIX [1] by *S. W. Draper*

This paper discusses the nature of expertise in UNIX, arguing that in certain senses of the word there are no experts. The consequences for interface design of revising the commonsense notion of expertise, particularly with respect to designing help facilities, are then discussed.

Users in the Real World by *D. Owen*

Based on the premise that people demonstrate a considerable degree of competence at formulating and achieving goals in the world, this paper seeks to identify and examine the relationship between the crucial characteristics of the real world and inherent or acquired human skills that support this competence, in order to improve the human computer interface. Aspects examined include a "naive physics" of computing and the reconstruction of propositionally held information.

Constructive Interaction: A Method for Studying User-Computer-User Interaction by *C. O'Malley, S. Draper, and M.S. Riley*

This paper describes a promising technique for studying human-machine interaction called Constructive Interaction (Miyake, 1982). It consists essentially of recording sessions with two participants who are discussing some topic which they do not fully understand. Miyake was interested in what was revealed about the underlying schemas of the participants and how new schemas can originate in an interaction between two people. We are interested in what this basic situation can offer for the study of HMI.

Formalizing Task Descriptions for Command Specification and Documentation by *P. Smolensky, M.L. Monty, and E. Conway*

The problem of formally describing computer tasks in terms of the input given and the output desired is considered. A feasibility study in the domain of printing suggests that *task attributes* provide a powerful language for such descriptions. It is argued that task description is important for moving the center of human-machine interface design away from the machine and toward the user.

---

1. UNIX is a trademark of Bell Laboratories. The comments in these papers refer to the 4.1 BSD version developed at the University of California, Berkeley.

Problems in Evaluation of Human-Computer Interfaces: A Case Study by *L.J. Bannon and C. O'Malley*

One of the most difficult aspects of interface design is evaluating new or changed features of an interface. This paper discusses methods of evaluation and design in the context of a program developed to assist users in getting quick access to information contained in the UNIX manual.

Planning Nets: A Framework for Analyzing User-Computer Interactions by *M.S. Riley and C. O'Malley*

During the course of interacting with a computer, a user has goals that correspond to tasks to be performed and must plan how to achieve those goals with the available commands. A framework for analyzing user goals, the mapping between those goals and available commands, and the factors influencing the success and efficiency of the resulting plans is presented. The implications of this analysis for the development of principles for improving user-computer interactions are discussed.

Activity Scripts by *A. Cypher*

A session with the computer can be organized around the activities of the user, rather than around the actions of the computer. A user-centered approach to grouping stereotypical sequences of commands into scripts or macros is discussed. This approach illustrates several issues in Human/Computer Interaction: *joint problem solving*, *tool/task mismatches*, and *visible effects*.

DESCRIBE: Environments for Specifying Commands and Retrieving Information by Elaboration by *S. Greenspan and P. Smolensky*

In communication between people, objects and events are principally referred to through *description*. This paper argues that the basic principles that make such reference by description possible can also be employed in communication between people and computers. A new type of operating system called *DESCRIBE* in which commands and files are referenced by description (as well as by name) is proposed.

Caveats on the Use of Expert Systems by *L.J. Bannon*

Recently we have witnessed a round of assertions and counter-assertions about the capabilities of applied AI, specifically in the area called "knowledge engineering," where scientists are involved in the building of so-called "expert systems" that are designed to mimic the performance of human experts in certain domains. Strong claims about the potential social benefits of such systems are being voiced, but this paper is concerned with the caveats.

Software Engineering for User Interfaces by *S. Draper and D.A. Norman*

The discipline of Software Engineering can be extended in a natural way to deal with the issues raised by a systematic approach to the design of human-machine interfaces. Two main points are made: that the user should be treated as part of the system being designed, and that projects should be organized to take account of the current (small) state of a priori knowledge about how to design interfaces.

STAGES AND LEVELS IN HUMAN-MACHINE INTERACTION

Donald A. Norman

The interaction between a person and a computer system involves four different stages of activities—intention, selection, execution, *and* evaluation—*each of which may occur at different levels of specification. Analysis of these stages and levels provides a useful way of looking at the issues of human-computer interaction.* [2]

My concern is with the overall process of interaction with the computer. I want to avoid an emphasis on detailed aspects of that interaction and ask about the nature of the interaction. Details are indeed important, but only once the proper conceptualization has been applied. Consider a simple situation. A user of a computer system is writing a paper and, in the process, decides that the appearance of the printed draft is not ideal: the paragraph indentation does not look proper. The user forms an intention: to correct the appearance of the paper. Now the problem is to satisfy this intention by translating it into the appropriate set of actions. The purpose of this paper is to examine some aspects of the interaction between a person and the computer system as the person attempts to satisfy the intention. The focus is derived from three observations:

1. When a person interacts with a computer, it is possible to identify four different stages of that interaction, each with different goals, different methods, and different needs (Norman, 1984).

2. Each of the known techniques for the interface has different virtues and different deficiencies. Any given method appears to lead to a series of tradeoffs. Moreover, the tradeoffs differ across the four stages of user interaction (Norman, 1983*a*).

3. Messages and interactions between user and machine can take place at a number of different levels. If the levels are not matched, confusion and misunderstanding can arise. Determining the appropriate level is a difficult task, often requiring some knowledge of the intentions of the user (Norman, 1981*a*, 1983*b*).

Let us start with a brief analysis of the stages.

---

2. The ideas discussed here result from interactions with members of the UCSD Human-Machine Interaction group. The manuscript has been submitted to the *International Journal of Man-Machine Studies*. Various sections of the paper have been presented at the SIGCHI Conference on Computer-Human Interaction (Norman, 1983*a*), the IFIPS First Conference on Human-Computer Interaction (Norman, 1984), and at the NSF Conference on Intelligent Interfaces (New Hampshire, 1983). Sondra Buffett and Edwina Rissland have provided helpful critiques of various drafts of the paper.

The Four Stages of User Activities

I define *intention* as the internal, mental characterization of the desired goal. Intention is the internal specification of action responsible for the initiation and guidance of the resulting activity. Although intentions are often conscious, they need not be. *Selection* is the stage of translating the intention into one of the actions possible at the moment. To go from intention to action, the person must review the available operations and select those that seem most auspicious for the satisfaction of the intention. Then, having mentally selected, the actual command sequences must be specified to the computer. The determination of a particular command or command sequence is *selection*; the act of entering the selections into the system is *execution*. *Intention* and *selection* are mental activities; *execution* involves the physical act of entering information into the computer. These activities do not complete the task. The results of the actions need *evaluation*, and that evaluation is used to direct further activity.

Thus, the full cycle of stages for a given interaction involves:

- Forming the intention;
- Selecting an action;
- Executing the action;
- Evaluating the outcome.

Perhaps the best way to understand the differences among the stages is to continue with our example. The intention is to improve the appearance of the printed version of the manuscript. This is a higher order statement that must get translated into more specific terms. Suppose that because it is the paragraph indentation that looks wrong, the user decides to switch to a "block paragraph" format—a format in which the initial line of a paragraph is not indented. We now have a second level of intention: call the main intention $intention_0$ and this new level $intention_1$. But even $intention_1$ is not sufficiently specific. Suppose the manuscript is being prepared by means of a traditional editor and run-off facility, so the manuscript contains formatting instructions that get interpreted at run-off time. One way to carry out the intention is to change the definition of the paragraph. Another way is to bypass the paragraph format specification and to substitute a blank line instead. There are several ways of carrying out each of these methods, but suppose that our user decides upon the latter technique, substituting for the paragraph specification *.pp*, the "skip a line" specification, *.sp*. This becomes $intention_2$.

Having formed $intention_2$, the next step is to *select* an appropriate set of actions to carry it out. This requires a set of text-editing commands, commands that find the appropriate location in the text that is to be changed (in this example, there are apt to be a rather large number of locations), then commands that change the *.pp* to *.sp*. There are several different ways of doing these operations. Thus, in the particular text editor that I happen to be using to write this paper (Berkeley UNIX *vi*), the following command sequence will do the operation *g/^.pp/s/pp/sp/*. [3] A more detailed analysis of the steps involved in making the selection would

3. As with many text editors, the command sequence is not particularly intelligible. The initial *g* signals that the command is to be performed "globally" to all occurrences of the string. The */^.pp/* is the string that is searched for in the text: a line that begins with ".pp." The remaining part of the line specifies the *substitute* command: substitute for the string "pp" the string "sp." Users of the *vi* text editor will recognize that even this description is slightly simplified. It should be clear that selecting this command string is a reasonably complex operation, requiring the setting of numerous sub-intentions and engaging in some problem-solving.

reveal that several more levels of intentions were involved. Eventually, however, a set of text-editing commands will be selected. We must take note of one more level of intention: the intention to execute the selected command sequence. Call this *intention*$_3$.

Having selected the command sequence, the next step is to *execute* the selection. In *vi*, this will require yet another action cycle to get the editor into the mode in which the substitute command will work properly, an action cycle that requires yet more levels of intentions, selection, execution, and evaluation. Finally, if all has gone well, the user has executed *intention*$_3$, and entered the command sequence into the system. Although *execution* has its own cycle of activities and sub-intentions, let us skip over them and assume that this stage has been performed properly.

This brings us to *evaluation*. Evaluation has to occur separately for each level of intention. First, it is necessary to check that the command sequence entered into the editor is the one intended. Then the manuscript text must be examined to make sure that *intention*$_3$ (the global change) got properly carried out: that all the *.pp* lines do indeed now say *.sp*. If they do, *intention*$_2$ (change pp to sp) has also been satisfied. Then *intention*$_1$ (change to block paragraph format) must be evaluated by means of yet another action cycle and another set of intentions. To see if the paragraphs come out in desired block-paragraph style, it is necessary to "run-off" the manuscript: this involves a new intention, *intention*$_{1A}$, and a new selection of commands. When all that is complete, the user can finally examine the printed page and determine whether *intention*$_1$ has been satisfied. If so, then the outcome can be evaluated with respect to *intention*$_0$ to determine whether the new format is a satisfactory improvement over the original.

Stages Are Approximations

Note that although it is useful to identify stages of user activity, the stages should be thought of as convenient approximations, not as well-defined, well-demarcated psychological states. People are not serial-processing mechanisms, they do not have well-defined stages of decision processes or action formation, and they often are not conscious of the reasons for their own actions. People are best viewed as highly-parallel processors with both conscious and subconscious processing, and with multiple factors continually interacting and competing to shape activity (see Norman, 1981*a*, *b*; Rumelhart & Norman, 1982). Nonetheless, the approximations used by this analysis may yield relevant and worthwhile results for the identification of important design considerations.

The Intention Stage

From the point of view of a system designer, there are two different aspects of intentions, each of which can be divided into two different concerns. The first aspect is the system's need (and ability) to know the intentions of the user, the second is the support that can be offered to the user to help form appropriate sub-intentions.

Knowing the user's intentions. Consider what the system might need to know about the user's intentions. There are two concerns here: (a) *what* needs to be known about a user's intentions, and (b) *how* it is possible for a computer system to get this information. The problem is made more complex because of the multiple-layers of intentions that exist, with any

reasonable task involving a fairly complex structure of intentions and sub-intentions. Still, for a system to provide useful guidance and feedback, it is going to need information about the user's higher-level intentions, both the overall, general intention and the sub-intention that is relevant at the moment (and perhaps the entire chain from the current sub-intention back to the highest level intention). Indeed, one could argue that all assistance (including help and error messages) requires input about the higher levels of user intentions in order to be maximally effective (see Johnson, Draper, & Soloway, 1983). The second concern, *how* a system can get the necessary information about the user's intentions, is the hard one. In some cases, the user can simply be asked. In others, it will be far more complicated. I expect that as we learn more about *what* the higher-levels of intention relevant to the task are, we will go a long way toward solving the *how* problem.

System support for sub-intention formation. There are usually two things a user needs to know in forming intentions: what is the current status of things?; what is possible, given the current status and system facilities? (Both of these points are also appropriate for other stages: the question "what is the current status?" is part of evaluation; the question "what is possible?" is part of selection.) We need to learn how to provide this information, at the appropriate level of sophistication for a given user at a given task, without intrusion.

The Selection Stage

Some intentions might map directly onto a single action, others might require a sequence of operations. In either case, the selection of an action sequence can require considerable knowledge on the part of the users. There are two aspects of selection. One is to figure out the method that is to be used in doing the task, the other to select which particular system commands are to be invoked. Consider how users decide what the options are in the selection process. How do they know the commands? There are four ways:

1. They could retrieve them from memory.

2. They might be be reminded, either by another person, the system, or a manual.

3. They might be able to construct or derive the possibilities.

4. They might have to be taught, either by another person, the system, or a manual.

In the first case, recall-memory is used to identify the desired item. In the second case, recognition-memory is used to identify the desired item from the list or description of the alternatives. In the third case, the user engages in problem solving, perhaps using analogy, perhaps eliminating possibilities. And in the fourth case, the user learns from some external source. This raises the issue of how the user knew that assistance was needed and how that assistance was then provided—a major theme of study in its own right.

Support for the selection stage comes principally from memory aids (manuals and various on-line support tools such as menus, help commands, and icons) that allow the user to determine the possible commands and their modes of operation, prerequisites, and implications. Selection can be enhanced by "workbenches" that collect together relevant files and software

support in one convenient location. Other methods of structuring groups of commands and files dependent upon the user's intentions need to be explored (for example, see Bannon, L., Cypher, A., Greenspan, S., & Monty, M.L., 1983).

The Execution Stage

Naming. There are two ways to specify an action to the computer: *naming* and *pointing.* Naming is the standard situation for most computer systems. The designer provides a command language and the users specify the desired action by naming it, usually typing the appropriate command language sequences. A speech input system would also be executing by naming. Execution by naming provides the designer with a number of issues to worry about. What is the form of the command language? How are the commands to be named, how are options to be specified? How are ill-formed sequences to be handled? How much support should be provided the user?

Most operating systems provide little or no support for intention, selection, or execution. The user is expected to have learned the appropriate commands. Then the execution is judged either to be legitimate (and therefore carried out) or erroneous (and an error message presented to the user).

It is quite possible for a system to provide considerable support for these stages, to provide information that tells not only the actions available, but also the exact procedure for executing them. This can be done with menus, perhaps abbreviated and restricted in content, so that they serve as reminders of the major actions available.

Pointing. Execution of an action by pointing means that the alternative actions are visually present and that the user physically moves some pointing device to indicate which of the displayed actions are to be performed. Although the prototypical "pointing" operation is to touch the desired alternative with a finger or other pointing device, the definition can be generalized to include any situation where a selection is made by moving an indicator to the desired location.

Note that a naming system requires two things: A place to point at and a means of pointing. We can separate these two. Moreover, as long as one needs a place to point at, it might as well be informative. Thus, the places serve as reminders to the selection stage when they consist of printed labels, lists, menus, or suggestive icons displayed on a terminal screen. But the places need not be informative: they might be unlabelled locations on the screen (or, in electronic devices, unlabelled—or illegible—panels).

Executing by naming often allows a large set of possible actions, hard to learn, but efficient in operation. Execution by pointing is restricted to those commands that can have a specified location. As a result, proponents of naming systems say they are more efficient: pointing requires sublevels. Proponents of pointing say they are easier to remember. One side emphasizes ease of execution, the other ease of selection.

The Evaluation Stage

Feedback is an integral part of evaluation, whether the operation has been completed successfully or whether it has failed. For full analysis, the user must know a number of things:

- What the previous state of the system was;
- What the intentions were;
- What action was executed;
- What happened;
- How the results correspond to the intentions and expectations;
- What alternatives are now possible.

The evaluation of an action depends upon the user's intentions for that action. In cases where the operation could not be performed, either because it wasn't executed properly, or because some necessary precondition was not satisfied, the user will probably maintain the same intention but attempt to correct whatever was inappropriate and then repeat the attempt. In cases where the operation was done, but with undesirable results, the user may need to "undo" the operation. In this case, repetition of the same action is not wanted.

One useful viewpoint is to think of all actions as iterations toward a goal. Ill-formed commands are to be thought of as partial descriptions of the proper command: they are approximations. This means that error messages and other forms of feedback must be sensitive to the intentions of the users, and, wherever possible, provide assistance that allows for modification of the execution and convergence upon the proper set of actions.

The user support relevant to each stage is summarized in Table 1.

Interface Aids

Menus in the Four Stages

One of the more common interface aids is a menu, implemented either as a set of verbal statements or as pictures ("icons"). It is useful to examine menus at this point for two reasons. First, the use of menus is often controversial, in part because their use requires trading the perceived value of the information provided by the menu for a loss of workspace and a time penalty (these tradeoffs are discussed in Norman, 1983a). Second, two different aspects of menus are often confounded. Menus can serve as a source of information for the intention and selection stages. In addition, they can also provide information, or even the mechanism, for the execution stage. That is, in execution by pointing, the menu or icon provides both information and a place to point. Unnecessary confusion arises when these rules of menus for selection are lumped with their roles for execution. Menus as sources of information for the intention and selection stages have one set of virtues and deficits; menus as mechanisms for the execution stage have another set of virtues and deficits. The point is that menus serve different purposes and have different tradeoff values for each stage: in part, the virtues for one stage are pitted against the deficits for another. The properties of menus can be summarized this way:

I. Menus are capable of providing information for *intention* and *selection* by:
   A. Presenting the user with a list of the alternatives;

Table 1

DESIGN IMPLICATIONS FOR THE STAGES OF USER ACTIVITIES	
STAGE	**TOOLS TO CONSIDER**
Forming the Intention	Structured Activities Workbenches Memory Aids Menus Explicit Statement of Intentions
Selecting the Action	Memory Aids Menus
Executing the Action	Ease of Specification Memory Aids Menus Naming (Command Languages) Pointing
Evaluating the Outcome	Sufficient Workspace Information Required Depends on Intentions Actions Are Iterations toward Goal Errors as Partial Descriptions Ease of Correction Messages Should Depend upon Intentions

B. Presenting descriptions and explanations of the alternatives.

II. Menus can aid in the *execution* stage:
   A. If execution is by *pointing*, menus can aid by:
      1. Providing a target to be pointed at.
   B. If execution is by *naming*, menus can aid by:
      1. Providing the user with an abbreviated execution name (such as the number of the menu line, a single letter, or a short abbreviation, usually mnemonic);
      2. Providing the user with the full command line (and arguments) that are to be used.

The first function of menus, providing information, is really their primary function. The information, explanations, and descriptions that they present are especially important in the stages of forming the intention and selecting the action. Note that this function can be done without any commitment to how execution is done. The second function of menus, aiding in execution, can be of equal use for execution by naming or pointing. Menus are especially useful when only a restricted number of alternatives is available, usually restricted to those described by the menu. Execution might be performed either by pointing at the menu items or by typing simplified command names (which are often so configured as to require only the typing of single characters).

Another major design decision is the question of how to get access to menus. The alternatives for menus are:

1. Always to be present in full form. Note that a set of labelled function keys can be thought of as a menu that is always present, with execution by pointing (i.e., depressing the appropriate key). In this sense, then, the panels of conventional instruments use a form of menus; the set of controls and range of possible actions are always visible. This option optimizes access to information at the expense of workspace.

2. Always to be present in a reduced form that allows the user to request the full menu. This option is a compromise position between the demands for information and workspace.

3. Not to be present unless requested by a special command or labelled key (e.g., "help") or by some other action (e.g., a "pop-up" menu called by depressing a button on a mouse). This option maximizes workspace at the expense of time and effort.

4. Available through a hierarchical or network structure, necessary when the menu size is large.

Note that fans of menus usually are those who weight heavily the information provided for intention and selection. Foes of menus usually are those who do not need assistance in these stages and who object to the loss of time and workspace during execution and evaluation. The differences come from differing needs at the different stages. Table 2 summarizes the effects of these issues on menu design.

Table 2

PROPERTIES OF MENUS		
VARIABLE	VIRTUES	DEFICITS
Information:	The more information presented in one display, the more detailed the explanations can be or the more alternatives can be presented, in either case improving the quality of advice offered the user.	More information increases times for searching, reading, and displaying, making it harder to find any given item, decreasing usability and user satisfaction.
Amount of workspace used:	The more workspace available for the menu, the more information can be displayed and the better it can be formatted, simplifying search and improving intelligibility.	The larger the percentage of the available space used, the more interference with other uses of that space.
Display of a large number of menu items:	Allows user to see a large percentage of the alternatives, aiding intention and selection stages and minimizing number of menus needed.	Slow to read, slow to display, uses large percentage of the available space.
Display of a small number of menu items:	Easy to read, quick to display, only a small percentage of the available space is required.	If number of alternatives is large, multiple menus must be provided. This can be slow and cumbersome.

Levels of Activity

The Problem of Levels

The existence of numerous levels of intentions leads to numerous difficulties. First, there can be a mismatch between the level at which the user wishes to express the intentions and the level that the system requires. Second, even apparently simple tasks can require considerable levels of intentions and sub-intentions, and the person's short-term memory may become overloaded, leading to confusion and error. [4] Finally, there can be difficulties in the evaluation stage, especially when the results are not as expected. Here the problem is to determine at what level the mismatch occurs. An example from a program on our system illustrates the problem. I wish to display the contents of a file on the screen. I execute the appropriate display program and it works fine. However, when I try to perform one of the options of the display program, the program collapses most ungracefully, and then displays this message on the screen: *longjmp botch: core dumped*. What is a "longjmp botch?" Why am I being told this? Of what use is this information to me?

The message was obviously written by a conscientious programmer who perhaps thought the situation would never arise, but that when it did, it would be important to tell the user. [5] One problem with this message is that it is presented at the lowest level of program execution whereas I am thinking at a fairly high level of intention: I want to change what material is on the screen and want it either to be done or to see a message telling me that it can't be done, in reasons relevant to my level of thought. "Longjmp botch" is not the level at which I am forming my intentions.

Remember the earlier example of attempting to reformat the paper. Suppose the end result is not satisfactory. Why not? The reason could lie at any level. Perhaps the run-off was not carried out properly; perhaps the change of *.pp* to *.sp* was not done properly; perhaps that change did not properly perform the "block paragraph" formatting; perhaps "block paragraph" is not what is required to satisfy $intention_0$. There are many places for error, many places where intentions could fail to be satisfied. If the operation were carried out manually, one step at a time, then it would be relatively easy to detect the place where the problem lies. But in many situations this is not possible: all we know is that the intention has not been satisfied. Many of us have experienced this problem, spending many hours "fixing" the wrong part of a program or task because we did not have the information required to judge the level at which the problem had occurred. The question, however, for the system designer is: what information is most useful for the user?

---

4. A number of "slips" of action occur for this reason, where the person loses track of the higher-order intention but continues to perform the actions associated with the lower-order ones. The result is to perform some action, only to wonder why the action is being done. When the lower-level actions are completed, there might no longer be any trace of the originating intention/action (an example from my collection: walk across the house to the kitchen, open the refrigerator door, then say "Why am I here?": Norman, 1981a).

5. It is from this and related experiences that I formulated the rule: programmers should never be allowed to communicate with the user. Good software design, I am convinced, can only come about when the part of the program that communicates with the user is encapsulated as a separate module of the program, written and maintained by an interface designer. Other parts of the program can communicate only with each other and with the interface module—most definitely *not* with the user. See Draper and Norman (1984).

The question is very difficult to answer. For the system programmer who is trying to debug the basic routines, the statement *longjmp botch* might be very useful—just the information that was needed. For me, it was worthless and frustrating. A statement like *System difficulties: forced to abort the display command* might have been much more effective for my purposes, but rather useless to the systems programmer. The problem is not that the error message is inappropriate; the problem is that sometimes it is appropriate, other times not.

One solution to the levels problem is to know the intention. If the program knew it was being used by a person who only wanted to see the files, it could make one set of responses. If it knew it was being used by someone trying to track down a problem, it could make another set. However, although knowing user intentions and levels often helps, it does not guarantee success. In my studies of human errors I have found numerous cases where knowledge of the intention would not help. Consider the following example:

> X leaves work and goes to his car in the parking lot. X inserts the key in the door, but the door will not open. X tries the key a second time: it still doesn't work. Puzzled, X reverses the key, then examines all the keys on the key ring to see if the correct key is being used. X then tries once more, walks around to the other door of the car to try yet again. In walking around, X notes that this is the incorrect car. X then goes to his own car and unlocks the door without difficulty.

I have a collection of examples like this, some involving cars, others apartments, offices, and homes. The common theme is that even though people may know their own intentions, they seem to tackle the problem at the lowest level, and then slowly, almost reluctantly, they pop up to higher levels of action and intention. If the door will not unlock, perhaps the key is not inserted properly. If it still won't work, perhaps it is the wrong key, and then, perhaps the door or the lock is stuck or broken. Determining that the attempt is being made at the wrong door seems difficult. Now perhaps the problem is the error messages are inappropriate: the door simply refuses to open. It would be better if the door could examine the key and respond "This key is for a different car." Can programs overcome this problem?

This paper is intended only to introduce the ideas that there are stages of activity, levels of intention, and tradeoffs among the solutions to the problems of human-user interaction. As the saying goes, more work is needed. But if that message is understood, then the paper is successful. My goal is to move the level of study of the human interface up, away from concentration upon the details of the interaction to consideration of the global issues.

References

Bannon, L., Cypher, A., Greenspan, S., & Monty, M.L. (1983). Evaluation and analysis of users' activity organization. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems*. New York: ACM.

Draper, S., & Norman, D. (1984). Software engineering for user interfaces. *Proceedings of the 7th International Conference on Software Engineering*. Orlando, FL.

Johnson, W. L., Draper, S., & Soloway, E. (1983). Classifying bugs is a tricky business. *Proceedings of the Seventh Annual NASA/Goddard Software Engineering Conference*. Baltimore.

Norman, D. A. (1981*a*). Categorization of action slips. *Psychological Review, 88*, 1-15.

Norman, D. A. (1981*b*) A psychologist views human processing: Human errors and other phenomena suggest processing mechanisms. *Proceedings of the International Joint Conference on Artificial Intelligence*, Vancouver, Canada.

Norman, D. A. (1983*a*). Design principles for human-computer interfaces. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human-Factors in Computing Systems*, New York: ACM.

Norman, D. A. (1983*b*). *On human error: Misdiagnosis and failure to detect the misdiagnosis*. Talk presented at the GA Technologies Inc. Workshop on Decision Processes in Operation Planning and Fault Diagnosis, La Jolla, CA.

Norman, D. A. (1984). Four stages of user activities. In B. Shackel, (Ed.), *INTERACT '84, First Conference on Human-Computer Interaction*. Amsterdam: North-Holland.

Rumelhart, D. E., & Norman, D. A. (1982). Simulating a skilled typist: A study of cognitive-motor performance. *Cognitive Science, 6*, 1-36.

THE NATURE OF EXPERTISE IN UNIX

Stephen W. Draper

This paper discusses the nature of expertise in UNIX, arguing that in certain senses of the word there are no experts. The consequences for interface design of revising the common-sense notion of expertise, particularly with respect to designing help facilities, are then discussed. [6]

Introduction

A frequently encountered common-sense view holds that in a computer system such as UNIX [7] there are experts and novices, experts being people who know more and can do more than novices. As novices learn, they gradually become expert. The supposition is that experts know things that novices do not, while the reverse is not true. A common suggestion built on this view is that a system can be tailored either for novices or for experts but not both, or that a system should have two modes, one each for novices and experts. An associated assumption is that novices need more help than experts and will make more use of any help facilities provided.

This paper will argue that the above view is wrong: that this apparently common-sense notion of "expert" does not provide an adequate analysis of the nature of expertise in systems like UNIX, and hence does not provide a sound basis for designing help facilities.

Command Usage Data

Over a period of 8 months data on the commands used on our system were collected: specifically how frequently each person used each command. The main measure extracted from this was each person's command vocabulary: the number of distinct commands that that person used at least once. The aspect of expertise reflected in this measure does not fit in with the above simplistic picture.

The Data Set

The data was collected over 8 months from a total of 94 people. They had about 570 commands available to them (the precise number fluctuates a little as new ones are added), of which only 394 were recorded as used at least once by at least one person. The largest vocabulary recorded for a single individual was 236. The data recorded the usage of our lab computer

---

6. Submitted to the First IFIP Conference on Human-Computer Interaction (London, September 1984).

7. UNIX is a trademark of Bell Laboratories. The comments in this paper refer to the 4.1 BSD version developed at the University of California, Berkeley.

whose user population includes programmers, psychology researchers (faculty, postdocs, and graduate students), and administrative staff. Most users use (some of) the word processing facilities, a minority use data analysis facilities, another overlapping minority use programming facilities. The computer ran 4.1 Berkeley UNIX, and in addition a substantial set of locally developed programs.

Its basis was the UNIX system accounting facility which records every process run and who ran it. Nightly this is collapsed, and for this purpose a cumulative record was created equivalent to a 2-D matrix of individuals versus commands with each cell recording the number of times that individual had used that command since the start of record-keeping.

This provides an easy method of mass data-gathering, but as we shall see there are a number of drawbacks inherent in this source of data which limit the conclusions that can be drawn from it. The first is that it records UNIX processes run, not user commands issued. Thus it records some processes that the user is unaware of having started (since they are called indirectly). This was largely corrected by a filter to eliminate those processes known to be called indirectly in almost all cases (e.g., the mail delivery program, as opposed to the program providing the user interface to the mail system) and also any processes not publicly available (e.g., private programs). This probably correctly eliminated over 90% of programs called indirectly at the cost of losing rare cases of individuals calling these programs directly.

A second consequence of recording processes not user commands is that this source of data misses all use of the 51 commands built into the shell (command interpreter) and not implemented as separate programs. There is reason to believe that this does not distort the trends in the data on which the arguments below depend, even though the built-in commands include some common commands, because the use of the built-in commands is typically tied to patterns involving recorded commands. For instance, the most common built-in command is probably "cd" (change directory). New users will not use this at first because they will not at first have created any subdirectories to move among. When they do begin to use it, they will also almost certainly begin to use the command "pwd" to show which directory they are currently in, and that is not built-in. Thus the overall trends and relative vocabulary sizes from this data are probably representative.

Another potential problem with estimating command vocabulary from such data is that a user might not use all the commands they know within the period observed. For instance, a researcher would only use data analysis tools in bursts at a particular stage of research. The long period of data collection (8 months) should however have compensated for this to a great extent.

Another problem is with data from the "superuser." There is a special privileged user ID (the "superuser"), which is used both to run system utilities and to give special privileges to one or two users for fixing up problems. Thus the system administrator (an "expert") runs part of the time under this ID. All data relating to this ID was discarded to avoid attributing automatic utilities to a person. The danger is that a considerable part of the expert's commands were also discarded. Since the expert however also runs under his own personal ID much of the time, it is probable that the size of his command vocabulary is accurately recorded even though the frequency of use of each command is underestimated.

Finally there is the problem of equating command vocabulary with expertise. Later in this paper the question of the nature of expertise is discussed. For now, it seems reasonable to take vocabulary as a good, though partial, indicator of expertise. It is especially appropriate in considering the help individuals may need which is so often information about the existence and name of commands.

Observations

The data make it clear that there were no experts on our system in the sense of individuals who used all the commands. There were a substantial number of commands never used by anyone (about 175). Furthermore, of the 394 commands used at least once by someone, the highest individual vocabulary observed was 236 or 60% of the total.

Next, the common-sense division of all users into novices and experts implies a bimodal distribution of expertise (here equated with the number of commands known) of which there is no trace in the data. Table 3 shows the number of users in each division of vocabulary size. There is a fairly smooth distribution of vocabulary size across our user population with perhaps a single slight peak in the lower half around a vocabulary size of about 45. The expectation of a bimodal distribution is of course a naive interpretation of the categorization, but note that the proposal to have a system with two levels of friendliness for the two levels of expertise is naive to just the same extent.

A more important observation comes from examining the extent to which one user's vocabulary overlaps another's. The common-sense model of expertise in which there is a single body of knowledge to be learned, and an expert simply knows more of it than a novice, would lead to the set of commands used by a user with a small total vocabulary (a novice) being a subset of the set used by a large vocabulary user. (We can call this the strict subset model of vocabulary knowledge.) This was not observed: on the contrary, each user's knowledge overlaps other users'. A useful image that conveys a picture of the situation is to imagine a Venn diagram of the sets of each user's commands looking like a flower with radiating petals: overlapping all others in the center, completely non-overlapping at the periphery, and with partial overlap of nearest neighbors in between. While users vary a lot in the size of their vocabulary (the petals are of various sizes), small users show the same pattern as large users: they use one or two commands used by almost all other users, one or two that no-one else uses, and in between commands with all degrees of shared usage. In fact, as far as I know, you cannot usefully plot the data as a Venn diagram because there is no way to plot the commands as points on the plane such that each user's set can be plotted as a simple closed curve (e.g., an ellipse). The important features of the distribution can however be perceived by considering the following.

If knowledge of commands were completely randomly distributed then the number of users per command would be approximately constant for all commands (a normal distribution around the mean of 20.4 users per command). Clearly this is not the case (see Table 4). On the other hand neither are commands acquired according to the strict subset model so that a large vocabulary always contains the commands in another user's smaller vocabulary. If that were so then the largest individual vocabulary (236) would be the same as the total combined

Table 3

Number of Users per Command	
Number of Users per Command	Number of Commands in this Division
100-91	2
90-81	8
80-71	12
70-61	13
60-51	12
50-41	24
40-31	24
30-21	34
20-11	70
10-1	195

vocabulary of the population (394).

In fact a substantial number of people know commands that no one else knows (whereas that would only be true of the largest vocabulary user in the strict subset model). There were 40 commands which only 1 user knew, and 18 individuals (rather than 1) each knew one or more of these idiosyncratic commands. Similarly, taking the group of commands used by 5 or fewer people, there were 128 such commands and 64 individuals with vocabularies spread evenly from the maximum (236) down to 25 each used one or more of them, but obviously the strict subset model would predict exactly 5 such individuals (those with the 5 largest vocabularies). In other words two thirds of the population used at least one "rare" command.

Thus the picture suggested by this data is that instead of there being a common body of known commands with users differing in how much of it they know, each user is an expert — or rather specialist — in a different corner of the system, even though the quantity of knowledge at least as measured by the number of commands certainly varies a lot. This is consistent with the familiar concept of specialists — that expertise is not concentrated in any one person but is distributed throughout the community, so that the doctor (medical expert) is not the expert on, say, the law.

This suggests that a more fruitful way of viewing a system's users is that they are all in essentially the same general situation of knowing some things and being ignorant of (and therefore sometimes needing help with) others. It follows that although a given individual in a particular context may need either complete, partial, or no help depending on their knowledge of that part of the system, the kind of help needed will not be a stable characteristic of that individual, i.e., that from the point of view of providing help, "expert" and "novice" are labels for contexts not for individuals. Thus any scheme, such as Schneider (1982) proposes, for classifying different levels of user skill, should be not applied uniformly for a given user but made specific to the command or context in question.

Help for Users

Scharer (1983) points out that users in general typically don't use manuals: at most they use one-sheet summaries, and they prefer to consult the local "expert." (We are now considering another aspect of the notion of expert — someone others consult.) Often this expert does not know the answer but does know how to use the documentation in order to extract the answer. This not only confirms the idea that experts do not know everything that even novices want to know, but suggests that the help facilities provided with the system (manuals) are more heavily used by experts than by novices. This is borne out by informal observation on our system — the resident expert is indeed by far the heaviest user of the printed manual, while novices seldom use it, and roughly speaking manual use is proportional to knowledge of the system and not inversely proportional as might have been expected. We also found this pattern in the data: we compared the number of calls each individual made to the on-line manual (the "man," "apropos," and "whatis" programs) with their command vocabulary: the correlation was positive with a value of 0.72.

It might be argued that this shows the effectiveness of the manuals: that those who use the on-line manual expand their vocabulary. However, in a given period only learners should have both high manual use and a high accumulated vocabulary; people who already had a large vocabulary at the start of the observation period would not be expected to show high manual

Table 4

Users' Vocabulary Sizes	
Vocabulary	Number of Users
240-231	1
230-221	0
220-231	0
210-201	0
200-191	1
190-181	2
180-171	2
170-161	5
160-151	1
150-141	5
140-131	4
130-121	2
120-111	6
110-101	5
100-91	4
90-81	6
80-71	5
70-61	8
60-51	10
50-41	4
40-31	10
30-21	7
20-11	3
10-1	3

use. The correlation suggests another cause at work, or alternatively that most high vocabulary users were "learners" in this period.

This pattern may in part be because the people whose job it is to solve problems others cannot solve, and to bring into effective local use new facilities which were hitherto non-existent, non-functioning, or unknown locally, are naturally those with the most knowledge; yet because their job is to tackle new things, naturally they need "help" from documentation. These are the people who in fact therefore need the most help from the system at least in the sense of new information of the kind traditionally enshrined in documentation: system designers should presumably therefore be tailoring a large part of the help facilities for them.

An Interpretation

The above should be enough to show that a more careful analysis is called for. In fact every case is different — a unique combination of task and user experience, where that experience has a number of relevant components that cannot be properly compressed onto a single dimension. These components include (i) the user's knowledge of using that system for that task, (ii) his knowledge of that task independent of that computer system, (iii) his general knowledge of that computer system, and (iv) his general knowledge of other similar systems. The kind of help most appropriate will vary with all these components.

The simple observations and data gathering described above thus lead to a quite different (though in retrospect perhaps not so surprising) view of user knowledge and expertise than the one based on the common-sense notion of an expert. It seems that in designing a system (and especially its help and documentation facilities) to match the expertise of its users, one should expect a user community, no matter what the overall expertise of its members, to contain users in all states of knowledge of any particular command or area, and furthermore that it will not be possible to predict a given user's expertise in one area on the basis of their overall knowledge of the system. All users will be experts in relation to some parts of the system, novices in relation to other areas they have never used, and intermediate elsewhere.

This is not to say that there is no sense at all in which individuals may acquire a general expertise in a system. Scharer's observations suggest that one important aspect of general expertise is the ability to extract information from the system and its documentation rather than relying on other people, as is supported by our data on the use of the on-line manual: such people have "manual dexterity" in Pat Wright's (1983) phrase. Another factor we may expect is a growing understanding of the system in general which makes it progressively easier to understand new things. This may be seen either in terms of mental models — of acquiring a more accurate knowledge of the underlying principles and components of the system — or of acquiring larger quantities of experience which simply raises the chance of any new problem being soluble by analogy to a previous one. In this generalized sense there is doubtless still a tenable concept of expertise which would carry some predictive power about the probability of a given user being able to solve some problem. Nevertheless it remains true that whether a user needs help depends on the combination of user and command and is not a property of the user alone. Thus it seems advisable for every command to provide support for three cases — for those who have never used it, those who have only used it occasionally, and those who are familiar with it — without making assumptions about their general level of knowledge.

With the above in mind, we can now argue that the pattern of expertise that was observed is what in fact you would expect in a large system (one with many commands), partly because different users have different tasks (specialization), and partly because for many tasks there will be more than one way to do it and different people pick different ones. Thus in a system with a small command set you would not expect to see it, but in other large systems you would. One such is the UNIX editor "vi," which has about 110 commands (cf. the approximately 600 UNIX commands at the shell level). Data was collected on its use from the same population, and a qualitatively similar pattern of command use was seen, especially when the use of compound commands is examined (e.g., "dw" for delete-word is a compound from the delete command, and the move-to-next-word command).

Conclusion

Command sets as large as in our UNIX system or in the vi editor are just too large for anyone to learn in their entirety, and there are not strong constraints common to all users on the order in which commands will be learned. Thus you should expect specialization not expertise at this level of knowing commands. At a more general level of knowing how to find things out about the system, the common-sense one-dimensional notion of expertise is more nearly applicable. This fits Scharer's experience; it means documentation is used most by experts. This corresponds with the use of library indexes: although many readers use them a bit, their heaviest users are librarians, and if a user has trouble they ask a librarian to help them with the index; they do not expect the librarian to know the answer from memory. Thus if someone says they "know UNIX" or spends some time "learning UNIX" you should not expect them to be familiar with the whole command set — probably not even half of it. You might however expect them to know how to find the answers to questions about the system. Bear in mind though that one of the most important things to know is whom to ask about a given question — who the local expert is on that area of UNIX — and that is not something usually taught or documented.

In summary: there are no experts in UNIX in the sense of people who know all the commands. While there are certainly some users with a larger command vocabulary than others, experts' real skill seems to lie less in familiarity with the whole command set than in discovery skills that allows them to find answers to the questions they cannot answer from memory. These skills include knowing how to get information by experimenting with the system, ability to use other sources of information such as source code, and knowing whom to ask. (All these skills, including the last, are observed in the highest degree in our local consultant, even though he is the one other people ask.)

There are two possible conclusions for designers from this: either write the documentation for experts and expect novices to get their help from local experts (i.e., accommodate to the status quo), or concentrate on making the documentation instantly useful to novices (i.e, usable without the acquisition of a lot of expertise in using the documentation) and perhaps give tutorials and other support for the methods of information acquisition listed above. Furthermore the designer should be ready to meet varying levels of expertise (say novice, intermediate, expert) when providing help, and not expect these levels to be properties of individuals across all commands, but to be, for a given individual, specific to at least a subject area and probably to particular commands.

References

Scharer, L. L. (1983). User training: Less is more. *Datamation, 29,* 175-182.

Schneider, M. L. (1982). Models for the design of static software user assistance. In Badre & Shneiderman (Eds.), *Directions in human computer interaction* (pp.137-148). Norwood, NJ: Ablex.

Wright, P. (1983). A user-oriented approach to creating computer documentation. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems* (pp.11-18). New York: ACM.

USERS IN THE REAL WORLD

David Owen

Based on the premise that people demonstrate a considerable degree of competence at formulating and achieving goals in the world, this paper seeks to identify and examine the relationship between the crucial characteristics of the real world and inherent or acquired human skills that support this competence, in order to improve the human computer interface. Aspects examined include a "naive physics" of computing and the reconstruction of propositionally held information. [8]

Introduction

Much of the current work on human-machine interface design starts with an analysis of difficulties users experience with specific existing software systems, e.g., operating systems and editors. This paper explores a complementary approach based on the premise that we demonstrate a considerable degree of competence at formulating and achieving goals in the world which does not readily transfer to a computing environment. The task is to identify and examine the relationship between the crucial characteristics of the real world and inherent or acquired human skills and motivations that support this apparent competence, with the aim of providing the same kind of support in a computing environment.

Two aspects are examined here. The first is concerned with the formulation and achievement of explicit goals, the kind which are generally inferable from people's actions and the second seeks to emphasize the importance of non-explicit meta goals, evidence of which is less apparent. An inherent danger in this kind of approach is that of limiting the exploitation of a new tool/medium to existing concepts, without exploring new ones. The intent here is to identify and acquire an understanding of issues at a level which does not evaporate in the face of new ways of structuring activities and is not bound to existing or anticipated hardware.

Formulating and Achieving Goals in the Real World

People somehow become acquainted with a range of tools/agents, how to mobilize them, and how to formulate goals in a way which relates to the means of achieving them. This sometimes involves perceiving existing situations, deciding on desirable changes, setting up preconditions for the changes, and then uttering the appropriate incantation to invoke the tool/agent (Norman, 1984). Alternatively and less precisely, partially understood current and desired states may motivate the heuristic choice of some strategy, which is believed to lead in roughly the right direction. In doing this people draw on a whole range of skills, memory aids, and in

---

8. The ideas result from interactions with the UCSD Human-Machine Interaction project, including Liam Bannon, Allen Cypher, Steve Draper, Donald Norman, Mary Riley, and Paul Smolensky, and with Brenda Laurel of Atari. Sondra Buffett and Nancy Casey helped in improving the presentation. Paper submitted to the First IFIP Conference on Human-Machine Interaction (London, September 1984).

particular, input to the total range of senses. Furthermore the skills seem well adapted to the cues and representational modes of the real world.

So what is it about these skills and the real world that facilitates this apparent competence, and to what extent can they be exploited equally well in an interface?

The 'Naive Physics' of Computing

People acquire a degree of knowledge of the "naive physics" of the world (Hayes, 1978); approximately how physical cause and effect mechanisms work, and there is a growing field of research concerned with establishing the primitives of this physics. DiSessa for example (diSessa, 1983) uses protocols to probe a naive subject's understanding of "sponginess," exposing the degree to which it is sufficient to explain some everyday phenomena and its limitations on confronting less common situations. Naive physics provides a basic understanding of what may or may not be possible, which is exploited in many situations.

- It is powerful in determining the plausibility of proposed combinations of tool-object-outcome. For example, it allows the user to infer the relative appropriateness of a sponge and a hammer for a task.
- It supports the innovative use of tools: a screwdriver can be used to open a tin of paint.
- It supports short cuts: having understood the essential procedures laid out in a recipe, many people will follow it only as is necessary to get the main effect.
- It is particularly important in being able to cope when things go wrong: when water does not emerge from the end of a hose pipe, most people are capable of generating some debugging strategies.

Similar situations are to be found in the computing domain but require a very different naive physics.

- In which contexts, for example, is it appropriate to use "rm" to remove something, and is it like removing a spot from a window or removing a chair from a room? Is "rm" or "mv" or mouse movement plus three clicks the appropriate way of changing the location of a word in a file and can it also be used, say, to delay the execution of some command for ten seconds?
- Some computing systems, like UNIX, positively encourage the innovative combination of their facilities.
- When lengthy, perhaps menu-based, interactive interfaces become irritating rather than supportive, it is desirable to short cut the prescribed procedure and issue just those commands which are relevant to the immediate task. To do this one must have an understanding of what is relevant.
- It is also the case that things occasionally go wrong, and one then needs some capacity to analyze why.

The question then arises as to what might be the important notions in a naive physics of computing. One reason for addressing this question for the real world is to be able to present new information in a way which facilitates the transition from novice to expert (diSessa, 1983). But in the computing domain there is the opportunity, at least to some extent, to tackle the problem in another way, that is to induce in the user a naive physics which will be more easily extendible. So an equally relevant question might be how best to help people acquire that knowledge.

The physics of computing. This deserves a longer examination than is possible here, but lets us consider one important aspect. The computing domain is one of symbols and so it seems the essential "physics" is at least in part that of symbols and their manipulation rather than of the objects which are represented. People appear to be familiar with symbolic representations and their limitations in the world. Only in cartoons do people knock nails in with a photograph of a hammer but it is accepted that a photograph gives a reasonably reliable idea of shape and color. Similarly, the fact that one can do things to symbolic forms which are not possible on the real thing is not unknown. It is reasonable to cut out the picture of the hammer and put it in a collage positioned over a picture of a nail, or declare that the saltcellar is Paris! What is not familiar is the degree to which the use of symbols can be exploited in new domains. It is possible, for example, to represent and manipulate tiny patches of a single letter in a font editor. Even the means of seeing the symbols is indirect; there is no absolute guarantee that what is evident on the screen is in any other sense there. [9] This substantial difference between everyday use of symbols and their use on a computer makes it hard to infer the similarity.

Hand in hand with the symbols explosion goes the capacity to break down hitherto elemental operations on symbols and, with almost limitless flexibility, combine them into new compound actions. A simple example apparent in studies of editors is the difficulty some people have with the process of inserting text in a line where there is apparently no space (Riley and O'Malley, 1984). It exposes the limitations of appealing to real world analogies without revealing the essential differences in the physics of the domains. Not only is it necessary for the user to grasp the possibility of decomposition and understand the new range of elemental operations but also to comprehend and accept a program designer's decision as to what constitutes an improved combination. [10] On the one hand this flexibility exceeds people's experience in that it is available in new domains. To return to the collage example, it is not normally possible to devise a tool which will, in one action, both make space for and position a picture of a thumb between the hammer and the nail. On the other hand, it falls short of people's experience of a domain in which they do make heavy use of symbol manipulation, namely that of natural language. In this domain one can convey the same information in many

---

9. In "insert" mode in the vi editor one can delete characters just entered without leaving that mode. The cursor moves back over them and they are "lost" to the editor but they are not removed from the screen.

10. Even a model requires some understanding of the nature of the objects involved which might limit the value of its use in isolation. It could be regarded as part of a bootstrapping process.

different ways and to a large extent rely on the hearer's shared access to the world. [11]

Inferring a naive physics. Two related conditions which seem to be important may be inferred from analyses of naive physics by diSessa (diSessa, 1983) and McCloskey (McCloskey et al., 1983). The latter describes and analyzes the commonly held misconception that an object that is carried by another moving object (a person running with a ball) will, if dropped, fall to the ground in a vertical straight line. Their hypothesis is that the misconception results from a misperception of events in the world caused by an inappropriate use of reference frames. However, it is clear that people can straighten out similar misconceptions without formal physics training if there is sufficient motivation. Spear fishermen for centuries have been able to cope with position distortions caused by the different refractive indices of air and water. For most people, the straight line misconception does not interfere with any common goal. But in the case of the fisherman, the absence of a fish on the end of the spear is unambiguously a failure of some understood and explicit intermediate goal towards eating. This would indicate that in the computing domain it is necessary to expose to the user the implicit sub-goals of a compound command, at least in one form of that command. For example, in the UNIX operating system there is no command which will simply create an empty file. Invoking an editor on a non-existent file will usually succeed in creating it and, amongst other things, will assign to it some protection status. However, the editor gives no indication that this is happening and in general there is little evidence that a complicated protection structure is being automatically developed until one attempts to transgress it. The intent here is not to reiterate the "more meaningful error messages" chestnut, but to suggest that the user be allowed to absorb the notion of (in this case) protection by making its presence as a sub-task apparent in non-error situations. [12]

In diSessa's (diSessa, 1983) probing of a subject's comprehension of elasticity, it was apparent that a major stumbling block to extending the understanding of how a tennis ball could bounce, to how a steel ball-bearing could bounce, was the fact that the elastic properties of a tennis ball are visible, whilst normally those of a ball-bearing are not. In fact the interviewer, in attempting to convince the subject of the similarity, points out that with strobe photography the squishiness of a ball-bearing could also be seen. The argument is again one for visibility. To fully exploit a person's capacity to infer a useful naive physics, as much as possible should be made apparent of both the nature of a procedure and the properties of the objects involved. [13]

None of the above is intended as an argument for making computing systems mimic their real world physical counterparts. On the contrary, the use of icons that look like filing cabinets is of questionable value if associated concepts are not supported.

---

11. Witness the difficulty people have with understanding how blindness or deafness affects the shared access assumptions. Blind people are often shouted at, and deaf people guided around obstacles.

12. An approach being tried at UCSD (Draper, unpublished) is an interactive version of a general file creation program which makes explicit the attributes associated with a file including its protection by requesting them explicitly and reporting impossible combinations of options with reasons for the inappropriateness.

13. The problem here is that few of the physical properties that distinguish objects in the world are inherent in computer objects (e.g., different file types). To arbitrarily assign them may lead to confusing inconsistency. More relevant properties are implied by the operations it is sensible to perform on them (e.g., print, execute). At the very least this difference could be made clearer. Some ways of doing this, short of operating system re-writes, are being explored at UCSD in a simple editor and a "notepad" system (Cypher, unpublished).

Representational Modes

There is a range of ways in which people can sense and subsequently represent the world: sight, sound, touch. Much of the state of the world is permanently apparent in different, often analogical, representational forms which people trade on heavily to distribute the load of comparing, remembering, and understanding. In the computing domain we are essentially reduced to one perceptual channel to sense the nature of, and interact with, the encapsulated world. One is not subjectively conscious of having to translate the softness of a sponge and the hardness of a nail to a different representational form to perceive the mismatch. For an equivalent operation on a machine one has to know or be given a quantized textual description of the relevant properties of an object and an agent in order to assess the appropriateness of their combination.

It can be seen as an inevitable consequence of the compression of the world into a limited space to be viewed propositionally through a small window. It is a powerful property of the computing medium that a large amount of information can be held in a small space, but that is incompatible with the space-taking analogical forms of representation which it is possible to make use of in the real world. In this respect computers and the world stand at different extremes in the tradeoff between compactness and multi-dimensional accessibility.

There is a challenge therefore to make a shift in this tradeoff, to relieve the user of the overwhelming emphasis on propositional forms of representation and reconstruct the information so held into more immediately accessible forms. Mice and larger bit-mapped terminals are undoubtedly useful, but they still represent a preoccupation with improving ways of interacting with a window on the encapsulated world rather than providing qualitatively different access mechanisms. An example is the notion of location in, and movement around, a directory structure. The power of analogical representations of these notions is only minimally exploited, but imagine a device attached to your workstation which had drawn out on it a plan of your directory structure, and that on that plan you could physically place and move a counter. The position of the counter indicates your current working directory, and moving the counter would be equivalent to issuing a change directory command via the keyboard. The user would not have to remember the commands or mouse clicks necessary to change directory, or the exact name of the directory. Recognizing the position of the counter in relation to the physical characteristics of the pad might make it unnecessary to read the label. Current screen based solutions require either a temporary change of screen to see your relative position in a graphic representation or force you to read and parse the directory name if it is permanently displayed. Even the direct action of moving the counter significantly changes the nature of the interface. It replaces moving a mouse, in order to move a pointer, to indicate the object on which some subsequently and similarly indicated action is to be performed. Of course advances in touch sensitive "flat" screens may allow a more sophisticated implementation than the one suggested above. The argument here is that there is a particular property of the world which people exploit, that future developments might explore.

Knowledge Acquisition

Having constructed an environment full of wonderful facilities, how are users going to find out about them? There are many sources of information for the motivated seeker both for a computer interface and in the rest of the world. However, much of the information about tools/agents in the world seems to be acquired at times when it is not being directly sought and may not be relevant to any immediate goal: billboards, T.V. advertising, or watching other people without necessarily being in an accepted student/teacher situation. One becomes aware of the facilities with almost no special effort, but knowledge of their existence may influence how some future goal is achieved or to the extent that goal generation is "tool availability" driven, whether they are even generated. Even if a facility is known and used, seeing it used or described by someone else can increase one's own understanding of it. There is little exploitation of this kind of dissemination of information in most computing systems and although many possibilities may spring to mind, the exact nature of this kind of knowledge acquisition, and the conditions under which it is acceptable, rather than irritating, are not obvious.

A pilot study addressing this issue has recently begun at UCSD. It currently takes the form of a program which users may call as a displacement activity, which will display on the screen a small piece of information about the local computing environment. It may also be called with an argument which serves to confine the information to a particular subject (e.g., vi, an editor and C, a programming language). To try to establish the kind of information it is useful to present, users are asked to indicate whether they find each instance of interest. [14] A simple editor is being developed which will allow people to contribute their own database entries, and it is intended to make the system self-maintaining by allowing user responses to censor the contributions. [15] Evaluation of the facilities will be based on a log of its use, comparison of the usage of several existing commands before and after its introduction, and user comments.

Hidden Goals and Explicit Goals

This section is an attempt to push the same examination of the real world for a contribution to a more amorphous aspect of what makes one interface better than another. There are several established ways of assessing the effectiveness of different interfaces, e.g., ease of learning, frequency of mistakes, effective throughput. But these may miss a range of important characteristics which contribute to a user's subjective feeling on using the interface. We appear to have a range of non-explicit emotional and aesthetic requirements whose satisfaction is rarely the main objective, but which influence the route taken to achieving a more concrete goal. These "hidden" goals are difficult to identify but their influence can be seen in some kinds of behavior. For example, the motives behind travelling to work by the scenic route one day and the highway another are hardly explained by "needing a change of view."

---

14. So far, surprisingly few people specify a subject, although those who do predictably have a higher "interest hit rate," (60% vs. 40% overall). Also whether people stop using the facility does not seem closely related to the success of their first few uses.

15. Although some pieces of information have been very popular, others not at all, it is not yet clear what the crucial differences are, and so some kind of self-censoring seems important.

Two extremes in the degree to which explicit goals or hidden goals are being satisfied occurs in the use of tools and toys respectively. The description of something as a tool (work) implies that one is most interested in the explicit outcome of its application, and in general less interested in the means by which it is achieved. For a toy or game although there is often an ostensibly desired outcome, like amassing gold pieces or scoring points, the main object of the exercise is to satisfy ill-defined hidden goals almost as a side effect of how the overt goal is achieved. This sweeping generalization serves to convey a sense of the distinction being used. From this I want to argue that the subjective degree of satisfaction afforded by two different ways of doing the same job, or using two functionally equivalent interfaces, reflects the degree to which they satisfy by side effects, the user's hidden goals.

There is little future in attempting a detailed analysis of the concept of "pleasure," and that is not what is being suggested here. The best that can be hoped for is some approximate classification of behavior patterns that the satisfaction of hidden goals apparently precipitates, and attempt to provide the opportunity for similar behavior in constructing an interface.

As an example consider the behavior exemplified by the use of alternate routes to work. It would indicate that in spite of the fact that one input device may be optimal for speed and efficiency for a particular application, it is important to provide functionally equivalent, sub-optimal alternatives merely for variety. For example, a system which relied heavily on a "mouse" or speech input without the provision for performing the same tasks via a keyboard or a data-pad would not facilitate what appears to be an important aspect of human behavior.

But what lies behind a craftsman's attachment to a particular tool, a golfer's loyalty to a particular driver, or a traveller's preference for a particular travel agent?

Is it that the qualities, capacities, and limitations of these extensions are thoroughly understood and trusted, that they will not spring surprises? If so then the argument made in an earlier section for making explicit the consequences of using a computing facility is reinforced.

Is it that the tool in the hand of the user acts as a procedural memory? In other words, the user no longer has to remember a detailed specification of what he wants to achieve, only that whatever it is, it can be achieved by his use of that tool. It allows the specification of a desired outcome, and the selection of the means for achieving it, to be collapsed into a single mental step. If so, then perhaps one program designer's way of carving up the space of possible activities and providing tools accordingly may be adequate.

Is it that the the tool and its implications for the organization of the domain are a function of the personality of the user and that every time they are used, the user's own identity or image is gratifyingly reinforced? (Although this may seem somewhat esoteric, it is a phenomenon which is exploited everyday in the advertising world.) If this is the case then there is little satisfaction in being forced to absorb the identity of the program designer, however objectively efficient it may prove to be. Users should be given every opportunity to modify tools to reflect their own conceptual framework.

Concluding Remarks

I have attempted to draw attention to some aspects of the way we interact with the world, and how they might be exploited to improve interaction with a computer. One direction indicated by some of these aspects is towards making the user more aware of how things are done and why they are done that way. A different view holds that the machine should be developed as an intelligent agent, which will infer a lot about the user's intentions and not trouble them with any details. These views are not mutually exclusive, but they do represent a difference in emphasis which there has not been space to discuss.

References

diSessa, A. A. (1983). Phenomenology and the evolution of intuition. In D. Gentner & A. L. Stevens (Eds.), *Mental models*. London: Erlbaum.

Hayes, P. J. (1978). The naive physics manifesto. In D. Michie (Ed.), *Expert systems in the microelectronic age*. Edinburgh: University Press.

McCloskey, M., Washburn, A., & Felch, L. (1983). Intuitive physics: The straight-down belief and its origin. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 9*, (4), 636-649.

Norman, D. A. (1984). Four stages of user activities. In B. Shackel (Ed.), *INTERACT '84, First Conference on Human-Computer Interaction*. Amsterdam: North-Holland.

Riley, M. S., & O'Malley, C. (1984). *Planning nets: A framework for studying user-computer interaction*. Manuscript submitted to the First IFIP Conference on Human-Computer Interaction (London, September 1984). Also included in this Techical Report.

CONSTRUCTIVE INTERACTION: A METHOD FOR STUDYING USER-COMPUTER-USER INTERACTION

Claire O'Malley, Stephen W. Draper, and Mary Riley

In this paper we describe a promising technique for studying human-machine interaction called Constructive Interaction. We discuss the merits of the technique in theory and in practice and describe briefly two kinds of pilot studies employing it. Constructive Interaction was developed by Naomi Miyake (Miyake, 1982). It consists essentially of recording sessions with two participants who are discussing some topic which they do not fully understand, in the hope of sharing their knowledge and arriving at a fuller understanding. Miyake was interested in what was revealed about the underlying schemas of the participants and how new schemas can originate in an interaction between two people. We are interested in what this basic situation can offer for the study of HMI. [16]

Introduction

The technique is a descendant of protocol studies in which subjects are asked to "think aloud," i.e., to report on their conscious thought processes while solving some problem. The potential problems with this technique include the doubtfulness of the connection between verbal reports and mental processes (cf. Ericsson & Simon, 1980) and whether having to make a verbal report changes the task significantly and thus invalidates any generalization of the findings to more naturalistic situations (i.e., the experimental situation is not ecologically valid). Both these objections hinge on the fact that the verbal activity is not intrinsic to the subject of study. In a two-person interaction the communication is not made for the investigator's benefit but for the other participant(s). In addition, even if a subject is poor at expressing her knowledge she is likely to persevere in trying to communicate until her partner does understand, while in traditional protocol analysis the investigator is left with the choice of intervening further with requests for clarification or of making inferences from the protocols.

Several researchers besides Miyake have taken protocols from interacting participants, e.g., Gentner & Norman (1977), Suchman (1983). In addition, Bainbridge (1979) discussed the use of verbal protocols, noting that it can be useful to record from two users who are working together to solve some problem, or from an experienced user guiding another, and commented: "Maximum communication of thoughts and knowledge, or admission of lack of knowledge, can then be natural aims rather than a source of embarrassment."

An important advantage of two-person studies is that the investigator need not be a complete expert in the topic discussed by the subjects (although some prior knowledge is necessary). It is possible to allow the subjects to explore a problem and to develop the solution.

---

16. Paper submitted to the First IFIP Conference on Human-Computer Interaction (London, September 1984).

Study of the transcripts may then allow the investigator not only to grasp their solution, but to extract the ways of expressing it that proved effective for the participants from among other less effective attempts. Furthermore in human-machine interface applications, subjects' choice of topic (if you allow them one) is itself revealing of where they perceive there to be problems in understanding a system, as one of our studies has shown.

Applying Constructive Interaction to human-machine interaction means studying what users tell each other, and this is an important but neglected topic. As Scharer points out (Scharer, 1983), and as casual observation shows too, a major, and often overwhelmingly predominant, source of information for users is what they can learn from other users (i.e., users don't read manuals, they ask other users). The complete human-machine interface, therefore, does not just consist of a user and the machine, but includes other users who support each other by supplying important ideas and information. This by itself is a strong motive for studying the exchange of information between users. This channel will probably always be important, but in the usual situation this is enhanced because any deficiencies in other channels (e.g., standard documentation) will be made up for in practice by asking other users. Thus, it is also vital to study this channel to detect deficiencies in other channels.

Miyake's original study (Miyake, 1982) concerned the problem of understanding how a sewing machine could make stitches. Two subjects were videotaped as they tried to develop an adequate theory, largely by verbal discussion but also using paper and pencil and other simple aids including eventually the sewing machine itself. The theory developed by subjects was found to involve several stages, each of which solved the problem posed by the previous stage. The recordings were analyzed to identify the few moments of crucial transition when a new stage was reached for one or other of the participants. Applying this to various aspects of the UNIX [17] user interface, we ran several pilot studies in which two participants were videotaped as they discussed some part of the interface.

Our studies involved different topics of discussion, and different mixes of participants in terms of their prior knowledge of the topic. Participants had the use of a terminal with access to the system as required, and sometimes made pencil and paper sketches, which were also recorded. The topics were only loosely determined beforehand, mainly by the participants themselves, and there was minimal intervention by the investigators.

A Study of Problem-Solving about System Concepts

The first study we shall discuss was directly comparable to Miyake's studies in that it addressed mainly a mental models issue. The topic was the UNIX C-shell (command interpreter) — in particular, the rules governing when variable values will get passed to subordinate processes. This is governed by a consistent model, but one which is not discussed in the existing documentation, at least in connection with variable values. The two participants were both conscious of having a fund of experiences which they had not succeeded in connecting by a coherent theory and were interested in trying to do so. They both knew the system moderately well but were not experts. The study consisted of two one-hour sessions, with two subjects using pencil and paper diagrams and experimenting with the system. The whole interaction, including the diagrams and the screen, were videotaped. The investigator was present but did not

17. UNIX is a trademark of Bell Laboratories. The comments in this paper refer to the 4.1 BSD version developed at the University of California, Berkeley.

intervene.

The first session revealed early on which system objects and concepts the participants already knew about. Both subjects knew that the command interpreter was the C-shell, were aware of *.cshrc* files (these initialize the state of a user's C-shell), and knew of the existence of two kinds of variable (C-shell variables, and environment variables), and that these have different inheritance properties. It also revealed what had appeared as problems to the subjects: e.g., the distinction between the two kinds of variable (the *shell* variable and the *environment* variable), and the rules governing their transmission.

However, it also revealed that the subjects were seeking different *kinds* of explanation, based on different kinds of system models. S1 had considerable programming experience in UNIX, and tried to derive a theory for the differences between the variables and their transmission from his knowledge of the *fork* and *exec* system primitives which he knew were fundamental to UNIX. About a third of this first session was taken up with this construction and S1's explanation of it to S2, aided by diagrams that they drew. The amount of talking was fairly evenly divided, and S2 clearly understood S1's explanations — at least at a surface level.

The subjects then tried an experiment by typing an *exec* command to the C-shell. (This was actually suggested by S2. This test is possible because the shell implements an *exec* command that directly reflects the underlying *exec* system call.) When you type "*exec command*," the "command" is executed normally but instead of then getting back the shell prompt, indicating that the shell is present and ready for the next command, you get the login prompt indicating that the shell has "died" and your login session finished. This reflects the fact that the shell "*exec*-ed" the command — replaced itself by the command — and shows by implication that normally a shell runs a command as a separate, specially created, process.

This was a crucial experiment for S1, in that it provided for him a direct confirmation and illustration of his understanding of the shell's operation in terms of the system's primitives. However, S2 did not find it at all illuminating despite an ensuing lengthy discussion between the two. Thus, this is an example of how a crucially informative observation for one subject — the subject with a model in which it fitted — may have no value to the other subject despite attempts at explanation.

S2 had a different conceptual model, which was based on his knowledge of other systems, and his attempts to understand UNIX were all attempts to relate UNIX to this prior conceptual model. Fortunately, S1 knew of a system corresponding sufficiently to S2's model, and subsequently set about constructing a model of UNIX that fitted that of S2. However that model — in which there is a single active controlling process that dominates all events in the system — is quite inappropriate for describing UNIX. Nevertheless, although S2's model was, in a formal sense, of no use for understanding UNIX, in practice it determined what questions he wanted answered, and conversely what observations were ignored because they were apparently unrelated to those questions. Thus for S2 it was important to understand what process is listening to a terminal before a user logs in, what happens to it when a successful login is performed, and how this relates to the shell with which a user interacts after login.

The session illustrates the importance of prior concepts on understanding. For S1, "understanding" meant relating observations to his knowledge of system primitives, while for S2 it meant relating them to his model of another system. The latter case especially shows that

prior theory determines the problems to be solved and what counts as an explanation even when it cannot provide the explanation. Consistent with this was the ending of the first session, where both subjects had extended their understanding by their internal criteria but realized that this had not helped them at all with one of their original questions: the relationship between the two kinds of variables in the shell.

The second session was almost entirely concerned with experimenting with the system in an attempt to resolve this question. The session ended after a relatively short time, and before the subjects felt they had fully resolved their questions, at least partly because of the somewhat confusing factors with which they were faced. The first of these was their use of two different shells (command interpreters). Our version of UNIX supports two alternative shells, and our subjects at this point called up both in turn. This had the virtue that they could examine differences which are potentially informative, but it also meant they were dealing with two rather different syntaxes. The second source of confusion, or at any rate complexity, is that the two different kinds of variables are handled not just by different commands in the C-shell, but by commands with a different syntax. In addition, different commands are needed to *display* as opposed to *set* environment variables (while the same command in two different forms is used for shell variables). Finally, the subjects used nested instances of shells (by calling a shell within a shell) as part of their exploration. Not surprisingly, they found it hard to keep track of what was happening, and of what conclusions might be drawn about the theories they were comparing.

Nevertheless, experimentation is a reasonable and informative way to discover things about the UNIX system, and our subjects made some useful discoveries before abandoning the session. This suggests that the shells should be improved in order to support this method of gaining understanding. One step towards this would be a simplification of the commands for setting, changing, and displaying variables. Another, suggested by the first session, would be to provide a shell version of the *fork* command to go with the *exec* command, so that partially informed users could explore the effect of these crucial system primitives.

The kind of conclusions for documentation that might be drawn from this study are that two introductions should be written for UNIX, one explaining the basic approach to process creation implemented by *fork* and *exec* and relating the operation of the shell to these; and another explaining the basic system entities, the control relationships between them, and the sequence of processes involved in a typical user session from before login through to logout. In general, a whole set of these conceptual narratives would be necessary, and while most users might be interested in all of them to some degree, different users would select different items as the essential one for giving them the feeling of understanding.

Tutorial Sessions for Novices

The tutorial studies differed from the study described above in that the participants were unequal in their knowledge about the system. The situation involved a novice user with very little prior experience with computers, being introduced to the system for the first time by someone who had considerable experience with the system. There was little insight into the process of understanding in the tutor, since she was not developing her ideas in the session. The session was, however, dominated by a conflict between what the tutor wanted to convey — which was a basic ability to login to the machine and read electronic mail — and the questions of the "student," which were largely driven by the screen display. In order to introduce

the user to the message system, the tutor had to spend over half the session explaining various aspects of the system as a result of the user's queries about what was happening on the screen, much of which was in fact unnecessary for learning how to perform the basic task of reading and sending mail.

One of the sources of confusion was that a new system had recently been installed for first time users, which gave information about the aliases being read in from their ".cshrc" and ".login" files, and gave them directions for where to find more information about changing their aliases. This information had in part been deliberately designed to make visible certain events and entities in the system (e.g., how variables were set upon login, and how to change these) in an attempt to counter the problem that, without them, users typically remained unaware of their existence, and the possibility of changing them, either for personal convenience or to fix problems that arose. However, the information was confusing for the novice, who had no knowledge of the entities to which the information referred, and it was frustrating to the tutor, who wanted to deal simply with the process of logging in and reading mail.

This reveals a conflict between designing the interface for the long term benefit of the learner or for ease of initial introduction. The tutor here clearly favored the latter, and found these aspects of the system a major impediment to what she wanted to teach. It also suggests an alternative approach which would be to set new users up so that upon login they enter the message system immediately, and are not confronted with the shell. The virtue of this is that they can learn to use one thing at a time undistracted, and that one thing (e.g., electronic mail) is a useful and meaningful activity complete in itself. A disadvantage is that, at least at first, a new user will see a system that is different from other people's, perhaps reducing the amount users can help each other and introducing the need to decide on transition points between versions of the system.

These sessions also revealed the importance of low-level protocols to the first time user. By this we mean the procedure (protocol) which is used in order to effect a smooth dialogue. For example, one protocol used in most user-system dialogues is that each command is followed by pressing the RETURN key. However, in our system, this is not a consistent protocol across different "environments." The protocol is appropriate for dialogue at the shell level, but not in the editor, where commands are executed as soon as they are specified. Furthermore, in both the editor and the message system, both kinds of protocol are present: so sometimes the RETURN key is required and sometimes not. These inconsistencies across different applications cease to be so much of a problem for the more experienced user, who can recognize different contexts and perform the correct procedures appropriately. However, these differences are not made explicit in most introductory manuals and tutorials. A majority of the pupils' questions, in all of these studies, were directed at this — a topic that the tutor (like written tutorials) did not seem to anticipate having to focus upon.

A number of other confusions about the system of the kind found in one-person protocol studies of novices (*cf.* Lewis & Mack, 1982; Mack, Lewis, & Carroll, 1982) also showed up. For instance subjects were confused about when they should take the initiative in interacting with the system, and about what a "prompt" was. Prompts can be viewed as turn-taking signals for the user/interface dialogue, but furthermore, depending on context, they indicate that a *particular* kind of response is required from the user. Some prompts may be interpreted as "ready" signals for the next command from the user, whereas other prompts might be interpreted as specific requests for information. In addition, prompts of different types may be used as

signals or reminders of the context or environment (e.g., the editor versus the shell). This kind of knowledge tended to be taken for granted by the tutor in our studies, but it also revealed an aspect which is not made explicit by written documentation.

These sessions were also interesting in revealing some of the expectations which users seemed to have about the results of their actions. For example, one subject was confused by the fact that, having quit the editor, the text still stayed on the screen. A similar example was where the subject had logged out and was confused when the screen did not clear.

Other problems were created by ambiguities in referring expressions: for example, what the word "next" refers to in the context of the message system. A subject thought that typing "next" would get her the second message, since she had already read the first one, however, she had then left the message system and had just come back to it.

Subjects also had problems with the idea that they could be in different "environments" within the same system. In fact, depending on their primary goal for using the system (i.e., text editing, or using the message system) they tended to assume that that was all there was to the system. In fact, there are at least three different "environments" in which new users might find themselves: the shell, the editor, and the message system.

An example of the kinds of problems that can arise due to these different environments is the following: the subject thought he was already in the editor, and forgot that he had to "call it up." Since this subject had only used the system for word processing, the editor was the only part of the system he was aware of. Another subject tried to type out a message from the shell level.

Conclusions

In this paper we have reported on some exploratory studies in applying the technique of Constructive Interaction to studies of interaction among users and between users and machines. There are three characteristics which distinguish Constructive Interaction from simple two-person studies. The first of these is that the participants should have comparable knowledge about the topic. Secondly, they should want to solve the same problem. Finally, the emphasis should be on understanding or developing concepts, as opposed to learning procedures. Both kinds of study are useful. The first study we described was closest to Miyake's in spirit since it involved two participants of approximately equal expertise in discussing a topic they had chosen and were both interested in understanding better. The choice of topic was informative, as were the partial solutions they reached. The tutorial studies afforded a different kind of information, and the two participants revealed different kinds of information from each other. The "pupil" showed clearly the problems a novice can have with the system and with a tutorial. The tutor revealed information about how they think beginners should be introduced — about what information is relevant to explain initially, and what should be left out. In studying tutorials given by people, clearly a two-person study is needed; in studying novices' problems with a system, a conventional one-person protocol study might do as well, but it is probably much easier to get a novice to articulate questions to a tutor — who is obliged to try and give a useful answer — than to "think aloud" in a way that benefits the investigator (but not the subject).

References

Bainbridge, L. (1979). Verbal reports as evidence of the process operator's knowledge. *International Journal of Man-Machine Studies, 11,* 411-436.

Ericsson, K. A., & Simon, H. A. (1980). Verbal reports as data. *Psychological Review, 87,* 215-251.

Gentner, D. R., & Norman, D. A. (1977). *The FLOW tutor: Schemas for tutoring* (Tech. Rep. No. 7702). La Jolla: University of California, San Diego, Center For Human Information Processing.

Lewis, C., & Mack, R. (1982). *The role of abduction in learning to use a computer system* (Tech. Rep. No. RC 9433 (#41620)). New York: IBM Thomas Watson Research Center.

Mack, R., Lewis, C., & Carroll, J. (1982). *Learning to use word processors: Problems and prospects* (Tech. Rep. No. RC 9712 (#42887)). New York: IBM Thomas Watson Research Center.

Miyake, N. (1982). *Constructive interaction* (Tech. Rep. No. 8206). La Jolla: University of California, San Diego, Center For Human Information Processing.

Scharer, L. L. (1983). User training: Less is more. *Datamation,* 29, 175-182.

Suchman, L. A. (1983, August). *The problem with human-machine interaction.* Paper presented at the annual meeting of the Society for the Study of Social Problems Theory Division/American Sociological Association, Detroit, MI.

FORMALIZING TASK DESCRIPTIONS FOR COMMAND SPECIFICATION AND DOCUMENTATION

Paul Smolensky, Melissa L. Monty, and Eileen Conway

We consider the problem of formally describing computer tasks not *in terms of procedures that will accomplish them but rather in terms of the input given and the output desired. A feasibility study in the domain of printing suggests that* task attributes *provide a powerful language for such descriptions. We describe the constraints such attributes must satisfy, and the procedure we used to design the printing attributes and test their usability. Applications to attribute-oriented interfaces and documentation are discussed. It is argued that task description is important for moving the center of human-machine interface design away from the machine and toward the user.* [18]

The goal of human-machine interface design is to maximize the effectiveness of a mapping between two worlds: the world of *tasks* users need to perform and the world of *tools* provided by the machine. Since, traditionally, designers have depended on users to adapt their task needs to the available tools, establishing a mapping that pays comparable attention to these two worlds would constitute significant progress in interface design. The advent of powerful computers means that tools can now adapt more to users' tasks. To take advantage of this, interface designers must deepen their understanding of the task world; this understanding is a prerequisite for making the design of human-machine systems less machine-centered and more user-centered.

Our sense of the term "task" must be distinguished from the sense it has acquired from "task analysis," a powerful tool for studying interfaces (Kieras & Polson, 1982; Bannon et al., 1983; Moran, 1983; Riley & O'Malley, 1984). Analyzing tasks has traditionally been taken to mean analyzing *the procedures used to perform tasks.* In our terminology, interface studies of this kind analyze the mapping between the *user's mental tools* and the *machine's tools* for performing tasks. By contrast, we are analyzing tasks in terms of transformations affected on objects *without considering the processes* ("tools") used to perform the transformation.

To develop a more formal understanding of the user's task world we are studying the limited domain of *printing tasks on a computer system.* This domain is rich and can be reasonably isolated from other computer tasks. Our investigation leads us to suggest that:

(a) tasks be described in a formal framework of *task attributes*;

(b) computer tools be redesigned using task attributes;

---

18. Paper submitted to the First IFIP Conference on Human-Computer Interaction (London, September 1984). This work is part of the research conducted by the Documentation Group of the Human Machine Interaction Project at the University of California, San Diego. Claire O'Malley made particularly important contributions to this work.

(c) documentation be redesigned using task attributes (even if attribute-oriented tools are not adopted).

In this paper, we first explain what is meant by task attributes, then discuss how we derived and tested our attributes for printing tasks and argue for the use of attributes in the design of interfaces and documentation. Our research is in its early stages; we are not describing a fully implemented system, but rather presenting an approach and reporting on some feasibility studies.

A Formal Framework for Task Description: Attributes

"Printing" refers to a large variety of tasks that differ in several minor and major respects. This diversity is reflected in the variety of hardware and software tools that have been developed for printing. Our UNIX computing environment, for instance, has over 30 printing commands; each command can be invoked with several flags that each modify the command's result. Command lines in which the output of one program becomes the input of another are often required. Selecting the appropriate tools and creating the correct command line to take a source document file and produce the desired output is not a trivial matter. To isolate this piece of the user's responsibility, *we have assumed that the source document file has already been appropriately edited*; any necessary formatting macros are assumed to be included in the source file. We shall see that certain problems arise from this way of narrowing the scope of study, because printing in UNIX is not divided cleanly between the editing and post-editing phases when a task-based rather than tool-based viewpoint is adopted. [19] Incorporating editing of the source file will be an important and challenging extension of our approach.

Given a source document file, there are thus many different printing tasks that can be performed with it. The *attributes* of printing tasks are the dimensions along which these different tasks can be distinguished. An individual task is specified by a value (or in some cases, a set of values) for each attribute. The attributes [20] provide coordinates for the space of all printing tasks that can be carried out on a given computer system.

Initially we imagined that a half dozen attributes would suffice for printing. It quickly became apparent, however, that several times this many would be needed to specify tasks with sufficient precision. We now have 20 attributes, and have not yet fully covered the array of printing tasks. Several of these attributes are shown in Figure 1. The attribute names we have used are printed in boldface type; beneath each attribute, in italics, are its possible values, with hierarchical organization imposed in some cases.

Design of task attributes, like most such design problems, is at this stage more an art than a science; much iterative improvement is required. However, there are a number of properties that constrain the set of attributes.

---

19. For a general analysis of document formatting systems and a summary of UNIX document formatting, see Furuta, Scofield & Shaw (1982).

20. For conciseness, the term "attributes" will often refer to the set of all attributes and values.

output
- *hard copy*
- *soft copy*

paper
- *separated pages*
  - *preprinted letterhead stationery*
  - *plain lgp paper* ...
- *continuous perforated pages*
  - *11" x 14"* ...

printing method
- *full character impact print (daisy wheel)*
- *electrostatic wet process print (laser printer)*
- *dot matrix impact print (decwriter)*
- *pen-scribed print (graphics plotter)*

formatting
- *none*
- *equations*
- *tables*
- *references*
- *text*
  - *csl macros*
  - *ms macros* ...

portion of file(s) printed
- *all*
- *page(s) _____ through _____* ...

headers
- *none*
- *date*
- *file name*
- *page number*
- *given in file* ...

columns
- *none*
- *several files printed on each page, (each file in its own column)*
- *one file printed on a page, in _____ columns*

• • •

Figure 1. Examples of printing attributes (bold) and their values (italics).

(1) Any printing task performable on the system must be describable using the values for the attributes.

(2) Any two distinguishable printing tasks must have different values on at least one attribute.

(3) Each attribute should measure a single conceptual dimension of the task.

(4) Values must be definable with reference *only* to the input and output of the task; no reference to processes (software) is allowed.

(5) Attributes and values must be comprehensible to users.

Implementing Attributes: A Feasibility Study

Procedure for Determining Attributes

Finding a set of attributes and values that would meet all the above constraints required many developmental stages. We began by enumerating the printing commands available on our research-laboratory UNIX computer system. We then organized the various commands along a few obvious dimensions like hard/soft copy and formatted/unformatted text. We asked several users which factors tended to determine their choice of printing command. Some users demanded sufficient left margin to permit mounting in a loose-leaf notebook, others required that page breaks not artificially interrupt source program listings. At this point it became clear that the variety of contexts present in our lab and the variety of personal preferences required a long list of attributes for users to specify the important features of their printing tasks. This led to a list of many detailed properties that distinguish between the various printing programs.

To see what concepts seemed most important for experts in the printing domain, the method of *constructive interaction* was used (Miyake, 1982; O'Malley, Draper & Riley, 1984). Two experienced system users/developers were videotaped as they together tried to organize the printing commands in ways they thought were most useful. This refined somewhat but mostly confirmed our list of properties of printing programs.

To ensure that our attributes covered a representative variety of tasks, we recorded all uses of the printing commands for several weeks. User command-histories were also consulted for the printing tasks they contained. The printing-command lines were collected and used 1) to generate values for our list of printing attributes and 2) for checking whether the constraints cited above were satisfied. When command lines were encountered that could not be described, new attributes and values were added; when two different command lines (e.g., differing in only one flag) could not be distinguished in terms of their attribute values, again new attributes or values were added, or old ones refined.

Eventually we settled on a set of 20 attributes and corresponding values that seemed to meet constraints (1) through (4) above, in the restricted area of hard-copy printing. Most conceptual difficulties came from the fourth constraint of process-independent definitions. Time and time again we found ourselves wanting to define attribute values through the program or device that did the job rather than the job itself. We got most embroiled in printing details in the area of character size and spacing. The easiest dimensions along which to differentiate alternatives tended to vary across devices and programs, and it was challenging to find dimen-

sions that worked in all cases. For instance, the point size of typeset print is most simply defined by the *height* of letters, while for the various print sizes of our dot-matrix terminal, only the *width* of letters was variable. However it is *precisely because* it took effort for us to unify the ways of thinking about tasks across tools that we feel our attributes have something to offer in making coherent sense of the world of printing tasks.

The most serious difficulties arose from assuming that formatting macros were already present in the source document file. As mentioned earlier, in our UNIX system a clean distinction is not made between aspects of printed documents that are determined *within* the source document file and those that are determined *outside* the source file at the time of printing. The point size of type, for example, is usually determined by appropriate typesetting commands within the source file, but this can be overridden or supplied in the command line; page headings are sometimes determined explicitly by formatting commands in the source file but sometimes implicitly by the selection of a printing command that automatically creates a heading. As a result, a possible value for several of the attributes is *"determined within the source file."* It would seem more elegant if a given attribute were either always determined within the source file or always determined outside the file, but that is not the case for our system.

Usability of Attributes

It remained to be seen whether criterion (5) was met: could users actually use our attributes for describing printing tasks? To assess this, we devised 7 hard-copy printing tasks that varied widely. Users familiar with our computer system to varying extents were shown a raw printout of the source document, and a hard copy that defined the "desired result." They were given a checklist with all possible values for all the attributes; their job was to check all the values that described the "desired result."

We concluded from this informal study that attributes provide a very useful mechanism for users to describe tasks. Users with an understanding of typesetting were able to use the attribute descriptions with no instruction; others quickly learned the meaning of the attributes when allowed to ask questions. Users did have considerable difficulty understanding the within/without source file distinction; some instruction here might have helped significantly. Like us, users had to think hardest about the attributes concerning character size and spacing; it is a level of detail that is rarely thought about with any comprehensiveness. However, when asked to address these matters and when given supporting documents to consult (with examples of different fonts, sizes, spacing, and so on), users did fairly well. They tended to be uncomfortable specifying attributes that do not consciously enter in their choice of printing commands. Several people made encouraging comments about the value of the approach, and most said they expanded their knowledge about the printing capabilities of our system.

A study was also done to explore which attributes users would want to specify when producing a document. Users were given a verbal description of a realistic task situation, a source file, and a sample of a hard copy suggesting what they *might* want to produce. They were given the attribute checklist and asked to indicate those values they wanted to specify. The usability of the attributes was comparable to that of the other study; in addition, users wanted to assign weights to various attributes, ignoring some altogether.

Extensibility of Attributes

It is important that the set of attributes and values be expandable to accommodate unforeseen future task capabilities of evolving computer systems. This serves as a sixth constraint on the attributes, but one that is impossible to rigorously test. After the formulation of our attributes, the capability to make a hard copy of a bitmapped display screen was added to our system. This was a fairly good challenge to our attributes; for the first time one could print something that was not a file. However it was straightforward to change the attribute **source document file** to **source document**, adding the values *file* _____ and *bitmapped screen*. We have yet to see any reason for doubting that attributes offer a language for describing tasks that is as easily expandable as any such language could be; in fact we suspect that the lack of explicit tool-dependence in the attributes enhances their ability to accommodate task expansion from new tools.

Applications of Attributes in System Design

Redesigning Printing Tools

Attributes provide a powerful set of primitives for precisely specifying the task a user wishes to perform. They can in principle be used as a new basis for issuing commands. In such an attribute-oriented computing environment, the user would simply specify values for relevant attributes, and the computer would perform the necessary actions.

An attribute-oriented environment would work something like this: the system designers would formulate a set of task attributes in the various task domains, e.g., printing. They would write a program that would take a collection of values for attributes, request values for necessary missing attributes, compute appropriate values for attributes the user didn't care to specify, and perform the task. Our experience with printing attributes leads us to feel that (at least in this domain) such a general tool is feasible. In addition to writing this printing program, the system designers would create a collection of printing-attribute prototypes. Each prototype would be a package of values for printing attributes that describes a frequently executed task, like formatting text and printing it on the laser printer, printing a file on the user's screen without interpreting formatting commands, etc. These attribute/value packages would be given names; the two examples just mentioned might be called *format* and *show*.

While *format* and *show* are ways of using the general printing tool, to new users they would be "printing commands." Documentation would state the values each attribute has for each "command." To go beyond one of the standard "commands," a user would be able to access the package of attribute values defining that "command," modify it, and save the modified package under a name that could then be used as a new "command."

Some combinations of values for our attributes are simply not possible to realize; the attributes are not truly independent in this sense. A facility to help users create feasible packages of attribute values could be based on a database of rules encoding the interdependence of feasible attribute values (e.g., "if output=soft-copy then paper=none"). Users would start by specifying values for the attributes most important to them, and as they did so the system would interactively guide them by spelling out the implications of their choices, soliciting further choices from feasible values for the remaining attributes.

Attributes can in fact be used not just for specifying commands, but also for accessing files; a proposal for a unified attribute-oriented interface is presented in Greenspan and Smolensky (1984).

Tool-Based Documentation

The attributes we have developed allow users to specify printing tasks within the existing capabilities of our laboratory computing system. Without redesigning the printing software in the manner described in the previous section, the knowledge about the printing task world contained in the attributes are extremely useful for *documenting* the existing printing tools (Kieras & Polson, 1982).

In O'Malley, et al. (1983), two types of documentation were found to be needed by users; we'll refer to them as *tool-based* and *task-based*. Tool-based documentation is designed for users who want information about a specific hardware or software tool, such as what the "m" flag for the *lprint* command does, or whether a daisy-wheel printer can move up and down half-lines. Task-based documentation is designed for users who have a task to perform and don't know what tools are needed or even whether the task can be done at all. The imbalance between respect paid to the tool- and task-worlds is nowhere more evident than in documentation, where task-based documentation is vastly under-represented relative to tool-based documentation. This is no surprise, for the people who design tools already have the knowledge required to write tool-based documentation; task-based documentation requires developing an understanding of the task world and a language for talking about it. Attributes provide such a language.

O'Malley, et al. (1983) found two distinct needs for tool-based documentation: *full explanation* and *quick reference*.

Full explanation. Full explanation encompasses the two forms of documentation usually called "tutorials" and "users manuals." Manuals are typically an alphabetical sequence of entries describing the tools available in the system; the descriptions typically assume the reader is familiar with the necessary concepts. These concepts are presumably explained in the tutorials.

Attributes are precisely-defined concepts that we suggest should be used in the full explanations of software and hardware tools comprising manuals. They offer a uniformity to tool descriptions that facilitates the user's task of assimilating the variety of tools offered by powerful systems. In addition, the uniform set of underlying concepts embodied in the attributes can be explained to users in a document analogous to a tutorial; this *attribute encyclopedia* will be discussed below.

Quick reference. Attributes also enable concise but precise summaries of what commands achieve. Figure 2 shows a portion of the summary for one of our local printing commands, **lprint**. All printing commands can be summarized using the same set of attributes, and the precise meaning of the terms used can be found in the attribute encyclopedia. Task and tool characteristics are comparably salient; in the corresponding summary developed for a quick-reference facility that did not use attributes (Bannon & O'Malley, 1984), tool characteristics like program flags are salient, while task characteristics are buried.

lprint {*options*} {*files*} | prints {files} on laser printer (lgp) in 8 point fixed-width type with pagination and header. Does not interpret formatting commands.

* *The {} brackets enclose items to be substituted for. Do not type the {}.* *

attribute	default value	revised value . . . set by . . .	option
output	hard copy		
paper	plain laser paper		
printing method	electrostatic wet process (lgp)		
type font	stick font		
formatting	no macros interpreted		
portion of files printed	all	page {*n*} through end	+{n}
header	date, file name, page numbers	no header	-t
		{*word*} (no blanks)	-h {*word*}
		{*string*} (blanks ok)	-h "{*string*}"
direction of printing	standard	sideways on page, 2 columns	-l
columns	none	each {*file*} in its own column	-m
		each {*file*} printed in {*n*} columns	-{*n*}

Figure 2. Use of attributes to summarize the local command **lprint**.

Task-Based Documentation

Using attributes, documents can be written that describe precisely the tasks users can perform, leaving the tools that perform them in the background.

Attribute encyclopedia. In Figure 3, the entries are organized so that the document as a whole can be used as a tutorial on printing. The index to the encyclopedia directs users to the appropriate entry to learn about an attribute, value, or other term used synonymously or in connection with an attribute.

An attribute encyclopedia has several advantages over conventional tutorials and manuals. Unlike most manuals, it can be approached without prior knowledge about printing concepts. Unlike most tutorials, it goes into complete depth about the matters discussed. Like manuals, it consists of a number of separate entries that can be used independently for reference purposes. And like tutorials, it has some overall structure so that it can serve as an overview of the printing domain. The encyclopedia brings together all the information relevant to one aspect of a task, including information that in traditional, tool-oriented documentation, would be scattered across many documents. The information organized by attributes in Figure 3 is culled from the many sundry traditional tool-based documents, a few of which are shown in Figure 4.

The attribute encyclopedia is an excellent vehicle for expanding users' printing repertoire, because unfamiliar attributes would be clearly visible as would unfamiliar values for familiar attributes. In compiling the encyclopedia, the system documenters have already done the difficult work of pulling together the relevant pieces of myriad tool-based documents into a task-based structure. The encyclopedia is particularly valuable because the concepts and terms it presents are precisely those used in the other forms of documentation.

Task-to-tool index. The attribute encyclopedia deals mostly with the task world, referring to hardware only to discuss certain attributes (e.g., printing method) and referring not at all to software. In fact the same encyclopedia could be used in the redesigned attribute-oriented environment described above. In the present system, there is need for an additional form of documentation, a *task-to-tool index* taking a user's specification of a task using attributes and pointing to the appropriate software tools. The task-to-tool index can be implemented at various levels of sophistication. Simplest would be an on-line or on-paper index in which users would look up values for individual attributes finding the names of all the programs capable of printing with that value for that attribute. The user or the computer would then try to find a single program, or a way of combining several programs, to achieve the entire package of attribute values. A somewhat more involved approach would rely on a large database of command lines, each with the task it performs completely described with attribute values. A user would give a set of attribute values, and the database would be searched to find command lines that matched as closely as possible. A more sophisticated on-line system would work interactively. As a user specified the desired values for attributes, the system would indicate which programs are possibilities, and guide subsequent choices by listing those values available with the possible programs.

For previously discussed types of documentation, we have argued that attributes offer improvements by giving a uniform language for describing tasks. Task-to-tool documentation would be an extremely valuable new form of documentation; it is simply impossible without

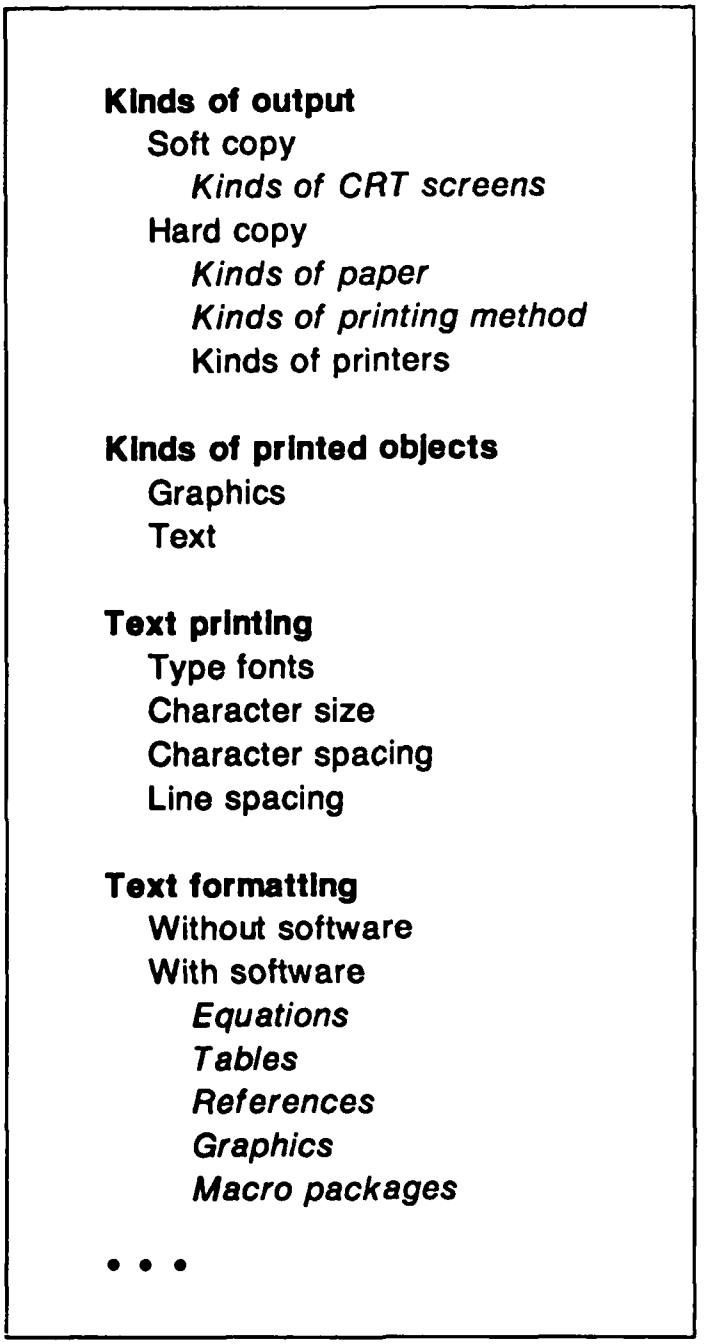


Figure 3. Task-based documentation: portion of contents of printing attribute encyclopedia.

Talking to the Computer with your new Tektronix 4010 Computer Display Terminal

Hewlett Packard 7221A Graphics Plotter Operating and Programming Manual

Laser Graphics Printer LGP-1 Technical Manual

UNIX for Beginners

User's Manual for the UNIX System

Typesetting Mathematics - User's Guide

tbl - A Program to Format Tables

Newgraph Tutorial

Sample Text and User's Manual for the csl Macros Package

Typing Documents on the UNIX System Using the -ms Macros with troff and nroff

• • •

Figure 4. Tool-based documents.

the kind of language provided by attributes.

Conclusion

In the domain of hard-copy printing, our investigations suggest that a set of about 20 *t*-tributes suffice for specifying the tasks that can be performed in a fairly powerful research laboratory computing environment. Users seem to find them useful ways of describing tasks. The attributes can be developed in a reasonable length of time. The attributes have potentially great utility for redesigning command specification, improving traditional forms of tool-based documentation, and permitting the development of powerful new kinds of task-based documentation.

References

Bannon, L., Cypher, A., Greenspan, S., & Monty, M. L. (1983). Evaluation and analysis of users' activity organization. In A. Janda (Ed.), *Proceedings of the CHI'83 Conference on Human Factors in Computing Systems*. New York: ACM.

Bannon, L., & O'Malley, C. (1984). *Problems in evaluation of human-computer interfaces: A case study*. Manuscript submitted to the First IFIP Conference on Human-Computer Interaction (London, September 1984). Also included in this Technical Report.

Furuta, R., Scofield, J., & Shaw, A. (1982). Document formatting systems: Survey, concepts, and issues. In J. Nievergelt, G. Coray, J. D. Nicoud, & A. C. Shaw (Eds.), *Document Preparation Systems*. Amsterdam: North-Holland.

Greenspan, S., & Smolensky, P. (1984). *DESCRIBE: Environments for specifying commands and retrieving information by elaboration*. Manuscript submitted to the First IFIP Conference on Human-Computer Interaction (London, September 1984). Also included in this Technical Report.

Kieras, D. E., & Polson, P. G. (in press). An approach to the formal analysis of user complexity. *International Journal of Man-Machine Studies*.

Miyake, N. (1982). *Constructive interaction* (Tech. Rep. No. 8206). La Jolla: University of California, San Diego, Center for Human Information Processing.

Moran, T. P. (1983). Getting into a system: External-internal task mapping analysis. In A. Janda (Ed.), *Proceedings of the CHI'83 Conference on Human Factors in Computing Systems*. New York: ACM.

O'Malley, C., Draper, S., & Riley, M. (1984). *Constructive interaction: A method for studying user-computer-user interaction*. Manuscript submitted to the First IFIP Conference on Human-Computer Interaction (London, September 1984). Also included in this Technical Report.

O'Malley, C., Smolensky, P., Bannon, L., Conway, E., Graham, J., Sokolov, J., & Monty, M. L. (1983). A proposal for user-centered system documentation. In A. Janda (Ed.), *Proceedings of the CHI'83 Conference on Human Factors in Computing Systems*. New York: ACM.

Riley, M., & O'Malley, C. (1984). *Planning nets: A framework for analyzing user-computer interactions*. Manuscript submitted to the First IFIP Conference on Human-Computer Interaction (London, September 1984). Also included in this Technical Report.

PROBLEMS IN EVALUATION OF HUMAN-COMPUTER INTERFACES: A CASE STUDY

Liam Bannon and Claire O'Malley

One of the most difficult aspects of interface design is evaluating new or changed features of an interface. In this paper we discuss methods of evaluation, their strengths and weaknesses, in the context of a program we developed to assist users in getting quick access to information contained in the UNIX [21] manual. We outline the problems encountered both in the design and the evaluation of this user interface. [22]

A basic tenet held by our research group is that the design and evaluation components of software development should be treated as a whole and not isolated from one another — evaluation should be considered from the outset of the design and built into the development of the system. We have tried to adhere to this principle in conducting our research. However, the task is not as easy as it may seem at first. Our experience in designing, implementing, and testing a small program resulted in several practical problems which are the subject of this paper.

Description of the Study

The study which we describe is part of our research on the development and use of system documentation in the Human-Machine Interaction project at UCSD. (cf. O'Malley et al., 1983). We have been examining how people at our Institute use the existing online documentation by monitoring their use of the programs, and by soliciting online feedback from users as they sought information in the manual. We found that about 35% of the use made of the online reference manual [23] was for what we refer to as "quick reference": users needed to be able to verify the name of a program, or check on flags, options, and syntax, without having to scan through extraneous material. In an attempt to meet this need, we developed a prototype online Quick Reference facility that contained only the correct syntax of the command, a list of possible options, and a brief explanation. This prototype system had a limited database consist-

---

21. UNIX is a trademark of Bell Laboratories. The comments in this paper refer to the 4.1 BSD version developed at the University of California, Berkeley.

22. Paper submitted to the First IFIP Conference on Human-Computer Interaction (London, September 1984).

23. The UNIX online manual contains separate entries for each program, which is accessed by typing the command *man* with the program name as the argument. This produces several screenfuls of text in a standardized format, with the name of the program, a short synopsis, a longer description of the program and how to use it, examples, and some diagnostic information.

ing of printing commands [24] which we felt would be both representative of the eventual system we had envisaged, and which would be immediately useful to our user population.

We were interested in determining whether the new facility would meet the quick reference need that we had already identified. Because our earlier data had been collected from examining the use of existing facilities in our Institute, we decided to evaluate the proposed new program within this same context, and assess how users changed their use of the original online reference manual after we introduced our new facility. This meant that we had to accept much less control over the possible variables than in a traditional experimental study, but it was a more appropriate method of evaluation at this stage in the development of the facility, as we were concerned about whether users found it of practical use in their everyday activities. A more controlled study would be appropriate for the purposes of debugging the specific display design, after we had determined the usefulness of this type of facility.

To evaluate the usefulness of the new program we decided to compare the frequency of use of *pref* with the use of *man* before and after its implementation by means of system accounting data. However, we also wanted more detailed information than simple usage data, in order to determine whether there were any problems with the facility, and what improvements should be made to it. One way to obtain this kind of information is from users themselves, by eliciting comments after each use of the program. However, earlier studies showed that many of our users complained that the request for online feedback was obtrusive, and interfered with the tasks they were performing. Therefore, in designing the evaluation tools for our new facility we included in our quick reference program a simple menu facility to allow easy user feedback on the usefulness of the information provided.

However, despite the fact that we went to some lengths to ensure that our methods of evaluation were carefully designed and conducted, we still had difficulty in giving a complete account of our results. In the following sections, we document some of these problems and try to relate them to more general issues in evaluation.

Design Aims and Evaluation Methods

Our main design considerations were the need for quick scanning of the material on the screen, brevity and clarity of the information, relevance and lack of redundancy, and clarity of syntax. In working on the design, we were very conscious of the necessity to make tradeoffs, and of the very limited information that was available to make such decisions. Some of our design objectives, such as reducing ambiguity and jargon, are not affected by the mix of users, but there were several aspects of the design where the demands of brevity and clarity pushed for different solutions, depending on the user population envisaged. The intended users of our new program encompassed a wide range — students, administrative staff, research faculty and staff — with varying degrees of knowledge about the UNIX system. Producing an interface that would be acceptable to such a wide variety of users was a difficult task, and led to a somewhat uneasy compromise.

As we have already discussed, we had two methods for collecting information on the usefulness of our facility: information concerning frequency of use from the system accounting data, and online comments from users.

---

24 The prototype system that we developed was called *pref* for "printing reference."

Online feedback. The online feedback was the most useful in *identifying common problems*, and it also served to suggest reasons for the patterns we found in the account data. The main drawback of this method was its intrusiveness on users, and we found that after a while users stopped providing comments. However, despite the potential annoyance to users, this method proved useful in identifying some of the problems encountered by our users.

System accounting information. The system accounting information, on the other hand, was useful in *identifying patterns of use*. It was especially useful in revealing the pattern which we characterize as "task-specific help," involving several successive calls to *man*, with different, but functionally related, arguments. However, there were some problems inherent in the use of the system accounting information. In investigating the frequency of use of the commands for which users were seeking help, it was difficult to determine exactly what users were doing, because the system accounting information also collects information on pre- and postprocessors called by the program, but not specified explicitly by the user. Distinguishing these data is difficult with the present system, since one has to use knowledge of the processors called, and timing information, to infer what was actually typed by the user. Thus the data had to be sifted "by hand" before any automated analysis could be conducted.

Given the problems outlined above, it seemed necessary to combine these two different means of collecting data: user comments indicated usage problems that were not obvious from examination of the system accounting information, while the latter data were more revealing of patterns of use. The more general point here is that the method of evaluation used should provide information on both the nature and range of problems, as well as their frequency of occurrence, and for this several methods of evaluation are required. On their own, these kinds of information provide only gross indications of problems, but in our case they did prove useful in identifying broad categories of help that users needed.

Results of Evaluation

System accounting information. One of our concerns in evaluating the system was whether our new program was more satisfactory than *man* for getting help on options and syntax. We reasoned that, if *pref* is useful, there should be some effect on the use of *man*. We had identified about 35% of the use of *man* with the need for quick reference, so we might expect about a 35% drop in *man* if that need was being fully served. We decided to compare the frequency of use of *man* for the period prior to implementing *pref*, with the use of *man* and *pref* following installation. [25] We did find a decrease in the use of *man* for *pref* users; however, there was also a decrease for those who did not use our facility. When we examined the frequency of use of the printing programs themselves we found a concomitant decrease, which could account for most of the overall decrease in the use of *man*. When we normalized the data to control for the decrease in use of printing commands, the difference in use of *man* between *pref* users and those who did not use *pref* was still large, so we can conclude that the use of *pref* had made a difference to the use of *man*.

---

25. One major problem was that *pref* was initially designed only to cater to the subset of commands which dealt with printing, so we had to select those calls to *man* that had the same arguments as those that *pref* covered, in order to do an accurate comparison.

Online feedback. The data described above gave us a general idea of the usefulness of *pref*, but we needed more detailed information. We had built into the design of the facility a means for obtaining this more detailed kind of evaluation which we felt would minimize disruption to users. Users were able to indicate their success or failure to obtain the information they wanted by a simple menu of commands (See Figure 5). In order to quit the program, the user types upper case Q(uit) if the facility is useful, and lower case q(uit) if it is not. [26] We also gave users the option of providing more detailed feedback, by typing c(omments), which puts them in the editor where they can type their comments and then return to the quick reference entry. Users are also able to get a more detailed explanation of the command from within the quick reference facility, by calling the regular online manual without having to quit the program, by using the command m(anual). They can also specify new arguments by using the "new" menu option. Users specify lower case n(ew) if the previous entry had been unhelpful, and upper case N(ew) if the previous entry had been helpful. Users can also get online help for the use of the facility, by typing h(elp).

We compared the use of upper and lower case quit commands, and found a significant difference in favor of the upper case (55%), indicating that users found the facility useful. We were concerned about whether users were simply perseverating by choosing to always type upper or lower case, so we examined the use of upper case N and lower case n, and found that there was no corresponding significant difference between these two, in fact there was a slight difference in the opposite direction. This ruled out simple case perseveration as an explanation of the result. We also considered the possibility that the use of *pref* and *man* might be for quite different sets of printing commands, which would therefore make a comparison between the two of rather limited usefulness. [27] However, when we correlated the use of the arguments to *pref* with the arguments to *man*, we found a significant positive correlation, indicating that the pattern was not appreciably different.

Another of the evaluation questions we had asked concerned what improvements could be made to the facility. The online feedback obtained from users provided useful suggestions about possible improvements: the comments indicated that they found the program a positive addition to existing facilities, and their suggestions were extremely helpful in further refining the facility at each design stage. For instance, the facility was changed to page rather than scroll after user comments indicated irritation with the scrolling.

Summary

In summary, our evaluation provided a number of measures that supported our hypothesis that a quick reference facility for system commands was needed. There was a drop in the use of the *man* command for printing programs after *pref* was introduced, and online user evaluations of *pref* after each usage were, on balance, favorable. What of longer-term evaluation? When we examined the patterns of use which emerged over the period between initial implementation of the facility and after it had been in use for a few months, there was a reduction in use of the prototype system. Our methods of evaluation supplied no obvious reasons for this.

---

26. The reason for using upper case to indicate success was to lessen the probability of response bias, since lower case is the normal form of a command, and it is easier to type.

27. The reasoning was that *pref* might just be supplementing specific inadequacies in some of the manual entries, as in some cases the information in the *pref* entries was more complete than that in *man*.

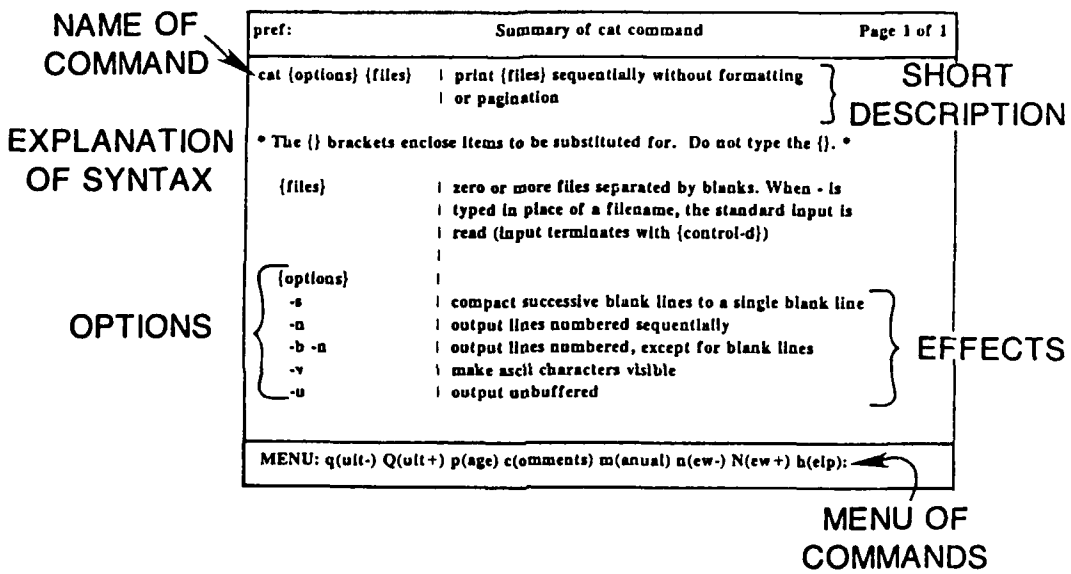


Figure 5. Example of *pref* screen display.

Conclusions

Some of the problems we encountered were due to the lack of control we had over the variables. Our study was not an experimental one in the tradition of laboratory studies, although we attempted to be as rigorous as possible in our data collection without intruding too much on our users. The main reason for choosing to test our system initially "in the field" was that we were concerned with the basic question of whether our program was serving its intended purpose in fulfilling a need which was not provided adequately by the existing documentation. The answer to this question could only be provided by studying the use of the system within the context of the existing facilities. Other questions concerning the details of the design, the format of the display, and so on, are more amenable to strictly controlled studies. However, we felt it important to test out our concept of the system before investing too heavily in details of a specific design.

We have discussed the importance of considering evaluation questions at the outset of the design. The need for iterative and piecemeal development, where prototypes are implemented and tested before the final version is developed, often leads to problems relating to the representability and scope of the system "pieces" that are initially chosen for implementation and evaluation. Testing of prototypes means that any modifications that have to be made cost less than with a complete version; however, the evaluation of the prototype system may not generalize to the proper context envisaged for the complete system. In our own case, one aspect of this problem of "modularity" which concerned us was whether or not the choice of domain in which to implement the prototype was a representative one. Our restriction of the quick reference facility to the printing commands was done for the sake of expediency, as we did not have the time nor the resources to build a database for the whole UNIX command set. The domain itself was certainly appropriate in that almost all of our users performed these tasks frequently, and often needed quick reference help. However, the database we implemented was still restricted in scope, and several users expressed dissatisfaction with the limited amount of information provided; users wanted information concerning programs which our prototype did not yet cover.

The reduction in use of our prototype over time is difficult to interpret. Some possible explanations include the following: some reduction in use of the facility is expected in the weeks following its introduction, as the novelty factor diminishes. Also, users may have forgotten about the facility, or they may have forgotten the name of the program — not an unlikely occurrence on a system with hundreds of commands. This latter problem is likely on our system because of the lack of support for accessing documentation (in general, the user needs to know the name of the program in order to access it.) Users may have memorized the relevant information in *pref*, thus reducing the need to actually call the facility.

Alternatively, rather than expending effort on trying to account for a slight reduction in usage of our new facility, perhaps we should consider a more basic question: what is an appropriate baseline level of use for such a facility? We had not really studied this issue prior to development, and we still do not have an answer to the question. The crucial issue about documentation facilities in general is not whether they are heavily used, but whether they satisfy the information needs of users on those occasions when they are used. (For example, our use of a dictionary may not be frequent, but can nevertheless be quite important.) This suggests that it may not be appropriate to rely on frequency of use information in the evaluation of the success or failure of a particular software facility.

References

O'Malley, C., Smolensky, P., Bannon, L., Conway, E., Graham, J., Sokolov, J., & Monty, M. (1983). A proposal for user centered system documentation. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems* (pp. 282-285). New York: ACM.

PLANNING NETS:
A FRAMEWORK FOR ANALYZING USER-COMPUTER INTERACTIONS

Mary Riley and Claire O'Malley

During the course of interacting with a computer, a user has goals that correspond to tasks to be performed and must plan how to achieve those goals with the available commands. We present a framework for analyzing user goals, the mapping between those goals and available commands, and the factors influencing the success and efficiency of the resulting plans. We discuss the implications of our analysis for the development of principles for improving user-computer interactions. [28]

Introduction

Our analyses so far have focused on learning and performance in the context of a single editor. However, an important objective of our approach is that these analyses achieve a level of description that will enable principles developed in this context to be extended to the instruction, design, and evaluation of editors in general, and eventually to other areas of the interface.

Theoretical Framework

The general form of our analysis is shown in Figure 6. The figure presents a typical planning episode in the form of a hierarchical goal structure — or *planning net*. At the higher levels of the planning net are global goals. Here the global goal is to edit a paper which in turn generates the additional goal to transpose two words. Since this goal does not correspond to an executable action, further goal specification and planning is required. *"Transpose two words"* is broken down into the subgoals *"delete word1"* and *"insert word1 after word2,"* which correspond to the actions of typing *"dw"* (delete word) and *"p"* (put), respectively.

Planning does not necessarily stop with the selection of the primary actions. Associated with actions are requisite conditions that must be taken into account in the planning process:

Prerequisites are conditions that must be satisfied before an action can be performed. Referring to the figure, the prerequisite of *"dw"* and *"p"* is that the cursor be at the appropriate location. Therefore additional goals are generated to ensure that those prerequisites are satisfied.

Consequences are the changes that result from performing an action. In the above example, the consequence of *"dw"* is that the word is deleted from the text and placed in a buffer. The consequence of *"p"* is to put the contents of the buffer at the location of the cursor.

---

28. Paper submitted to the First IFIP Conference on Human-Computer Interaction (London, September 1984).

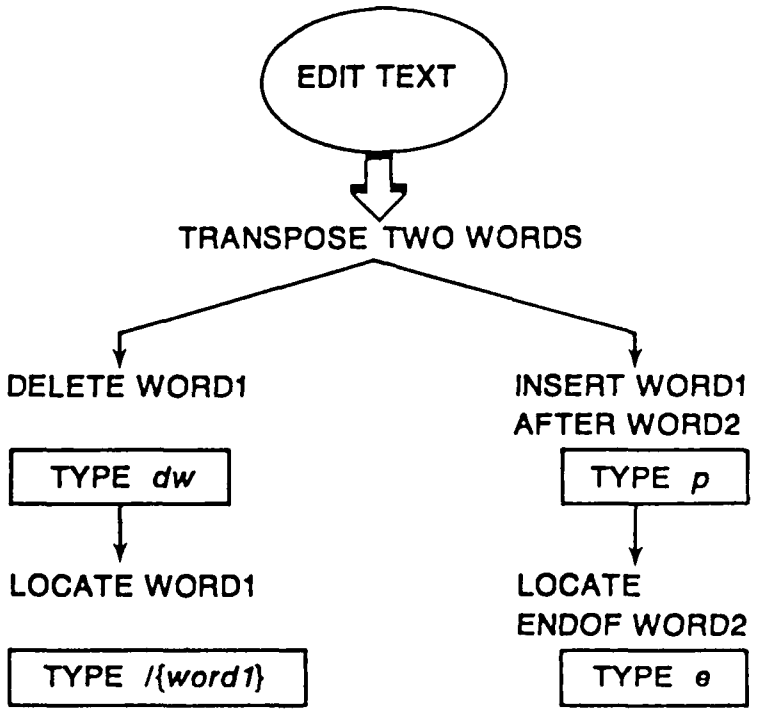


Figure 6. Planning net for the task *Transpose two words*.

These consequences define the *order* in which *"dw"* and *"p"* must be executed and, furthermore, place restrictions on interleaving plans. For example, other commands, such as "insert," also have the consequence of changing the contents of the buffer — if one of these commands were executed between *"dw"* and *"p,"* the consequence of *"p"* would be different (in this case *"p"* would be inserted as text).

Finally, some commands also have *postrequisites* — conditions that must be satisfied after performing an action. For example, the action of inserting text must be followed by pressing the ESCAPE key, to return to command mode.

Figure 7 shows an expanded version of the planning net for this example.

An important component in determining the success and efficiency of a planning episode like the one above is the mapping between the user's *mental model* of a command (the user's representation of how a command works), and the *conceptual model* of a command (how a command actually works). Furthermore, the likelihood that the user's mental model will correspond to the conceptual model is to a large extent a function of the *system image* — the feedback presented to the user before, during, and after the command is executed. (See Norman, 1983, for a more complete discussion).

The importance of the mapping between a user's mental model of a command and the conceptual model has been emphasized in several recent analyses (e.g., Card, Moran, & Newell, 1983; Kieras & Polson, 1982; Moran, 1983; Roberts & Moran, 1983; Young, 1983). However, the role of the *system image* has not been systematically distinguished from the conceptual model in these analyses. In our analysis we emphasize this distinction, showing how the feedback explicitly presented to users — especially in the learning phase — accounts for a large number of users' errors and misconceptions, independent of the command's conceptual model.

Empirical Study

Preliminary support for the usefulness of this framework has come from an empirical study of new users learning to use a text editor for the first time. The text editor used in this study was the UNIX [29] screen editor *"vi."*

Procedure

Subjects were six undergraduates who had never used a word-processor before, and who had minimal experience with computers in general. Subjects were studied individually approximately twice a week for a total of 4-5 sessions, each session lasting one hour. A typical session involved having subjects pace themselves through a written tutorial in the presence of an experimenter trained in taking protocol observations. Subjects were encouraged to think aloud while reading through the tutorial and performing the exercises. Audio, video, and keystroke information were recorded for each session and for the test that followed the instruction.

---

29. "Vi" is a screen oriented (visual display), command driven editor, based on "ex."

GOAL	GOAL	PREREQ	ACTION	CONSEQUENCE	
				system image	conceptual model
	Transpose 2 words	Locate word1			
SUBGOALa	Locate word1		Type "/{word1}"	Cursor at beginning of word	Cursor at beginning of word
SUBGOALb	Select delete operation		Type "d"		Delete operation selected
SUBGOALc	Mark range of operation		Type "w"	word1 deleted	Text deleted and placed in buffer
SUBGOALd	Replace word	Locate end of word2			
SUBGOALe	Locate end of word2		Type "e"	Cursor at end of word	Cursor at end of word
SUBGOALd	Replace word1		Type "p"	Word1 replaced	Word1 replaced from buffer

Figure 7. Expanded version of a *planning net*.

In the test phase, subjects were given a file to edit, which they were told contained several mistakes. They were also given a printout showing what the final version should look like. They were instructed to work through the file, taking as long as they liked, and to correct each of the mistakes they found. The corrections consisted of various core editing tasks (cf. Roberts & Moran, 1983) that required applying basic text editing operations (insert, delete, replace, transpose, or merge) to basic text objects (characters, words, lines, paragraphs). Subjects were also given a quick reference sheet, which contained a list of the basic editing commands which they had learned. (This was to ensure that we were not simply testing subjects' memory for the names of the commands.) Subjects were instructed to think aloud, telling the investigator what it was they were planning to do in solving the problems. The test lasted until the subject had completed all the tasks, or until one hour was up.

Results

Our initial analyses have focused on subjects' test performance, relying mainly on protocol transcripts from the audio-visual tapes, rather than the keystroke data. Overall results showed that subjects were correct on only 60% of the editing tasks, in spite of the fact they had practiced all the commands previously and had the list of commands available at all times. We grouped subjects' errors into three major categories, reflecting the stage in the planning process at which the errors occurred. These categories correspond closely to Norman's analysis of the stages involved in users' activities (cf. Norman, 1984).

The first category includes errors made during the formation of goals, or what Norman refers to as *intentions*. Approximately 15% of errors fell into this category. The second category includes errors made in the *selection* and/or *execution* of actions to achieve the specified intentions. Approximately 58% of errors fell into this category. In the third category we included errors that resulted from an *incorrect evaluation* of the outcome of performing an action. About 27% of the errors were of this kind.

These categories oversimplify what is really a *complex* and *interactive* planning process, and the errors we discuss reflect this. For example, many errors in forming intentions or in selecting actions were clearly the result of errors in evaluation from previous cycles of activity. Nevertheless, the categories are useful for identifying the stages of user activity that are not well supported by the system, and for suggesting specific changes to improve either the interface itself, or the instructional material.

Errors in the Formation of Intentions

Subjects often revealed very vaguely specified plans or intentions. We characterize this kind of error as a "fuzzy plan." (This finding is similar to what Lewis & Mack call "abduction" — Lewis & Mack, 1982). This captures the fact that a general intention is formed but there is no specification of that intention beyond this stage, and subjects can find no executable action that corresponds to that intention.

Example: The task was to insert a line of text above the first line, on which the cursor was positioned. The subject used the command *"i,"* correctly, in order to enter insert mode. She then typed the text she wanted, but realized, when she got near the end of the line, that the

"old" text [30] had to be put on the next line. This could have been achieved by typing RETURN but this did not occur to the subject. She pressed the ESCAPE key to leave insert mode, then tried to think of a way to get the "old" text onto the next line. She could only come up with a very vague (or "fuzzy") plan for achieving this. She seemed to have the plan of copying the text that was on the current line onto the next line (a generalization from the commands *"yank"* and *"put,"* as a way of copying text):

> S: *I know you can copy buffer, right? Or can I just delete it then add? I know there's some way we can erase it, then tell it to go somewhere else. Then push a button, and everything will be back.*

Other examples of problems at the intention stage included inefficient, or overspecified, plans. Even though subjects performed correctly on 60% of the tasks, about half of their correct responses were counted as "inefficient" plans. In other words, subjects tended to overspecify their intentions, so that they were operating with very primitive commands, rather than with the compound commands which would have achieved the same solution more efficiently. (This is consistent with findings from, e.g., Polley & Williges, 1982, and Robertson & Black, 1983).

Errors in the Selection and Execution of Actions

There were three main types of errors in this category: errors in predicting the scope of a command's consequence, errors in syntax, and errors in selecting text objects. An example of each of these errors is given below:

Example (Scope Error): The task was to delete to the end of the line, including the punctuation. There were five words on the line, and the subject typed *"d5w"* (delete five words). However, she did not realize that the text object *"w"* does not include punctuation, so she had an extra task of deleting the punctuation.

Example (Syntax Error): The task was to replace two words with three words. The subject forgot how to specify the object to the *substitute* command. He was confused about the syntax of the command, and gave the argument as the number of spaces for the "new" text (three words), rather than the number of spaces of the "old" text (two words).

Example (Text Object Error): The task was to delete to the end of the line. The subject typed *"ds,"* for "delete sentence" — generalizing from *"dw,"* for "delete word." The correct command, however, is *"dd"* (which is of course inconsistent).

An important feature of each of the examples in this category is that the fact an error has been made is immediately reflected by the system image — text intended for deletion remains on the screen (first example), text intended to remain on the screen is deleted (second example), or the system gives audible feedback when a text object is incorrectly specified (third example). As a result of the immediate feedback, many of these errors were corrected or the sub-

---

30 As "new" text is inserted in front of text that already exists, the "old" text is pushed along in front of the cursor

ject was able to ask for help. The next category also includes errors in selection, but the fact that an error has been made is not immediately reflected by the system image and therefore is not easily evaluated and corrected.

Errors in Evaluation of Actions

The success and efficiency of the subjects' plans was to a large extent a function of the mapping between the "conceptual model" of a command and the "system image." Difficulties arose when the system image failed to reflect important information about the *prerequisites* for selecting a command or about the *consequences* of executing a command.

(i) *Prerequisites:* There were three main types of errors involving prerequisites — the user either neglected to take into account a necessary prerequisite of an action, had an unnecessary prerequisite, or had a wrong prerequisite.

Example (Violation of Prerequisite): The subject's goal was to search for a pattern (achieved by preceding the string with *"/"*). However, she forgot to type the search command, thus the prerequisite of the action was violated, and this was not noticed by the subject (even though the system image revealed it). The error was compounded because the next two characters typed were "la." The problem was that *"a"* is the "append" command, which results in insert mode. This was not noticed by the subject — in fact the first time she noticed the error was when the next character "w" was echoed on the screen. This was an example of an error resulting from neglecting to take into account a necessary prerequisite of an action. In this case the subject was not aware of the mode she was in. The example illustrates how errors in evaluation can still occur even where the system provides the appropriate feedback. It also illustrates how errors can be compounded.

Example (Unnecessary Prerequisite): An example of an unnecessary prerequisite occurred when a subject thought that she had to be at the end of the line before giving the command *"dd"* to delete the line. (The cursor may be anywhere on the line in this case.)

Example (Wrong Prerequisite): Finally, an example of wrong prerequisites involved a subject who was aware of what mode she was in (she was typing in text) but chose a command whose prerequisite was that of being in command mode.

(ii) *Consequences:* In some cases a single action has more than one consequence, only one of which may be visible to the user. As a result, subjects often associated an action with only one of its consequences.

Example: The subject's goal was to delete a character at the end of a line. She chose the command *"A,"* in order to move the cursor to the end of the line. However, this also put her in insert mode, which she did not know, and could not evaluate because the consequence of being in insert mode was not made visible. In this case the error was both in selecting the wrong command as a result of learning only a partial consequence of the command *"A,"* and in evaluation, since the consequence of typing *"A"* was invisible.

Example: Another example was where the subject typed *"O"* to get a new line and then typed *"a"* to get into insert mode. The error in this case also resulted from the subject learning a partial consequence of a command: typing *"O"* does open up a space, but it also results in insert mode, but not realizing this, the subject then typed the command to enter insert mode. This error therefore also resulted from not being able to evaluate the consequences of an action since they were invisible, and from the subject learning only a partial consequence of the command *"O."*

Another example is where the wrong action is associated with a consequence, as a result of *delayed* consequences:

Example: Since the consequence of backspacing while in insert mode was delayed until after ESCAPE was pressed, the subject thought that it was ESCAPE that deleted the text, whereas it was the compound of backspace and ESCAPE that performed the action. Delayed consequences, therefore, caused the error of associating the most recent consequence with the most recent action. In this case, ESCAPE was a *postrequisite* for the action of erasing while in insert mode.

Subjects were also confused when the *intermediate* consequences of performing an action appeared to violate other goals: for example, the action of typing text while in insert mode has the consequence of typing over existing text until a special key (ESCAPE) is pressed to terminate the input mode. Again, in this case, the pressing of ESCAPE is a postrequisite for insertion of text.

In summary, we have identified some of the difficulties experienced by users in learning how to use a text editor, and we have related these difficulties to specific stages in the formation and execution of plans. In the next section we discuss the implications of our analysis for improving user-computer interactions. Our focus is mainly on the problems in evaluation, since they highlight the importance of the system image.

Implications

Intention errors: The errors in the intention category revealed that novices sometimes have problems in mapping their general plans or high level goals into executable actions. At other times they overspecify their goals into very primitive units. Our analysis does not provide any specific recommendations about what might be the right level at which to implement operations that would more directly map onto users' intentions. More research is needed to determine this.

Selection/Execution errors: The errors found in the selection/execution stage imply that instructions should make more explicit such things as the scope of a command, and the rules for generating commands (for example cross-product rules). Moreover, such rules should be consistent.

Evaluation errors: More direct implications for improving the interface come from the analysis of errors occurring at the evaluation stage. Difficulties in evaluation occurred when the system image failed to reflect important information about the prerequisites for selecting a command or about the consequences of executing a command. The direct suggestion for improving the interface is to make this information visible to the user. For example, many of the subjects' problems involved either not knowing which mode they were in before executing an action; knowing the current mode, but selecting a command which required being in another mode; or not realizing that an action resulted in a mode change. Here the implications for improving design are that an explicit indication of mode change should be provided, and furthermore, that any such indication should be salient to the user.

One way to make prerequisites more salient is to have error messages explicitly reflect which prerequisites are being violated instead of, for example, simply giving audible feedback to indicate that the command cannot be executed.

Other difficulties in evaluation occurred because subjects only learned the consequences that were made visible and failed to acquire those that were left invisible. Again, this suggests that the consequences of actions (for example, changes in mode, the contents of the buffer) should be visible. Making things visible not only gives users explicit feedback, but also encourages the development of a more coherent model, allowing users to predict, explain, and evaluate the behavior of the system.

Our analysis also suggests that not only should consequences be made visible, but to be associated with the correct action, they must be made visible immediately after performing the action (or at least before another command is executed).

Summary and Conclusions

We have suggested in this paper that difficulties in learning to use a text editor may be accounted for in terms of specific mappings and mismappings between the conceptual structure of a command, how that conceptual structure is reflected by the system image, and how users interpret that system image in terms of their mental models. These analyses suggest certain hypotheses about the knowledge required to generate efficient plans, how this relates to users' initial knowledge, and possible ways of helping users acquire more skilled levels of performance. Further theoretical and empirical work is required to test and extend our hypotheses. Nevertheless, results of this exploratory study support our idea that a planning framework is useful as a basis for developing general principles for instructing, designing, and evaluating features of an interface.

References

Card, S. K., Moran, T. P., & Newell, A. (1983). *The psychology of human-computer interaction.* Hillsdale, NJ: Erlbaum Associates.

Folley, L., & Williges, R. (1982). User models of text editing command languages. *Proceedings of the Conference on Human Factors in Computer Systems.* Gaithersburg, MD.

Kieras, D. E., & Polson, P. G. (in press). An approach to the formal analysis of user complexity. *International Journal of Man-Machine Studies.*

Lewis, C., & Mack, R. (1982). *The role of abduction in learning to use a computer system* (Tech. Rep. No. RC 9433 (#41620)). New York: IBM Thomas Watson Research Center.

Moran, T. P. (1983). Getting into a system: External-internal task mapping analysis. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems.* New York: ACM.

Norman, D. A. (1983). Some observations on mental models. In D. Gentner & A. L. Stevens (Eds.), *Mental models.* Hillsdale, NJ: Erlbaum Associates.

Norman, D. A. (1984). Four stages of user activities. In B. Shackel (Ed.), *INTERACT '84, First Conference on Human-Computer Interaction.* Amsterdam: North-Holland.

Roberts, T. L., & Moran, T. P. (1983). The evaluation of text editors: Methodology and empirical results. *Communications of the ACM, 26.*

Robertson, S., & Black, J. (1983). Planning units in text editing behavior. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems.* New York: ACM.

Young, R. M. (1983). Surrogates and mappings: Two kinds of conceptual models for interactive devices. In D. Gentner & A.L. Stevens (Eds.), *Mental models.* Hillsdale, NJ: Erlbaum Associates.

ACTIVITY SCRIPTS

Allen Cypher

A session with the computer can be organized around the **activities** *of the user, rather than around the* **actions** *of the computer. A user-centered approach to grouping stereotypical sequences of commands into scripts or macros is discussed. This approach illustrates several issues in Human/Computer Interaction: joint problem solving, tool/task mismatches, and visible effects.*

The study of activity scripts is part of a larger project which is concerned with organizing the multiple activities that a user engages in on the computer. These are not computer-centered activities like using an editor or an electronic mail facility; rather, they are user-centered activities like "preparing a paper" (which entails using an editor, a text-processor, and a file-handler) or "learning how to kill a job" (which may entail asking a colleague and consulting an on-line manual). Several issues in human-computer interaction arise in the context of activity scripts: *joint problem solving*, *tool/task mismatches*, and *visible effects*. Activity scripts deal with these three issues via *commands to the user*, *crossing program boundaries*, and *stepwise programming*, respectively. Activity scripts use the computer to reduce the burden on the user's short-term memory by keeping track of information and performing actions that would otherwise occupy the user's time and effort.

A convenient way to understand activity scripts is to compare the two activities of (a) issuing commands to the operating system, and (b) writing a line of a program. These two activities are quite different on current systems. System commands are typed one at a time, and each command is executed immediately after it is typed in. In contrast, programs are created as a whole, and the lines are only executed later, in sequence. The reason for this is clear: most of the things that we want an operating system to do can be accomplished in a single step, whereas programming is almost by definition the process of grouping together a series of commands to perform an action which cannot be accomplished in a single step. This at least is the conventional view of commands and of programs. But in fact, there are many cases where we want to group together sequences of commands to the operating system. And there are many cases where we would like to execute the lines of a program one at a time. Since these two activities are very similar from the user's perspective, it seems useful to attempt to provide a uniform means for performing them.

We would like to counter the notion that entering commands is not programming, and likewise to counter the notion that programs must be composed in chunks prior to executing them. Our work to date has been concerned with the implications of the former statement. Future work will explore the latter issue of applying "stepwise programming" to conventional programming languages.

Pursuing the notion that typical interactions with the computer system constitute programs of some sort has lead to several interesting ideas about human-computer interaction.

First, we view both the user and the computer as resources which can be used to perform an activity. This means that activity scripts will include some steps which the computer is to perform, and other steps which the user is to perform. This idea came from our empirical studies of command sequences. As an example, a user might ask the computer to display a list of current processes and their associated ID numbers. In the next command, the user will then issue a command to kill one of the jobs, which requires specification of the associated ID number. This sequence of commands could ideally be written as a program, except that it is quite difficult to write the code which searches for the job name and then returns the associated ID number. Nonetheless, that task is quite simple for a person to perform, and so the program is simple to write if we include a *command to the user*; a command which the user is to carry out. This is the essence of *joint problem solving*.

Second, viewing activities from the user's perspective leads to many activities which require the use of several different programs. This means that many sequences of commands will *cross program boundaries*. The fact that boundaries are crossed implies that the tools available to the user do not coincide with the user's conceptualization of the task. That is, there is a *tool/task mismatch*. If the user is permitted to create an activity script which crosses these program boundaries, that script will henceforth serve as a tool which coincides with the user's image of the task. The activity script smoothes over the seams between the different programs.

Third, we intend for users to create activity scripts simply by entering commands one at a time, as is customary. For instance, a user may execute several commands which search through the files in the current directory, locate those containing a particular string of text, and then move these files to another directory. If, sometime after this activity is performed, the user decides that this particular activity may have to be performed again in the future, that sequence of commands can be taken from the "history list" of previous commands and gathered into a script. This is the basis of *stepwise programming*. Of course, it is unlikely that the future use will involve the same search string or the same directories. A facility is therefore provided which allows the user to specify words or phrases in the script which are to be *variablized*: converted into variables. In this way, programs are written by performing the sequence of commands once, and then later deciding how to generalize the sequence. Since each command is actually executed during the first-use phase, the consequences of each command are *immediately visible effects*.

DESCRIBE:
ENVIRONMENTS FOR SPECIFYING COMMANDS AND RETRIEVING INFORMATION BY ELABORATION

Steven Greenspan and Paul Smolensky

In communication between people, objects and events are principally referred to through description. *This paper argues that the basic principles that make such reference by description possible can also be employed in communication between people and computers. A new type of operating system called* DESCRIBE *in which commands and files are referenced by description (as well as by name) is proposed.* [31]

Many operating systems do not offer users any way of retrieving files or specifying commands other than by name. For users who have no idea what a file or command name is, such systems offer no systematic help except structured exhaustive search (such as through a file hierarchy).

There is an attractive alternative to reference by name that is exploited to a tremendous extent in natural language: reference by description. In order to simulate the advantages of reference by description within a system of reference by name, the concept of "name" has sometimes been grotesquely distorted. [32] For example, a typical complete UNIX [33] filename, */csl/paul/pdp/monte/asynch/init.o* has crammed into it (from left to right) information about the file's disk location, creator, research project, research subproject, algorithm variant, contents (*init* = "initialization routines"), and type (*o* = "object code"). The file "name" has become an idiosyncratically and unsystematically encoded description containing information that is of value to users only if they possess a fair amount of idiosyncratic knowledge and of almost no use to the machine.

As an alternative, we propose a system called DESCRIBE [34] that systematically keeps track of this kind of descriptive information in forms usable by both machine and user, thereby fa-

---

31. This research was supported by a grant from the System Development Foundation, by contract N00014-79-C-0323, NR 667-437 with the Personnel and Training Research Programs of the Office of Naval Research, and by Grant PHS MH 14268 to the Center for Human Information Processing from the National Institute of Mental Health.

32. For a related discussion, see Norman (1983).

33. UNIX is a trademark of Bell Laboratories. The comments in this paper refer to the 4.1BSD version developed at the University of California, Berkeley.

34. DESCRIBE is a recursive acronym for the title of the paper.

cilitating communication. [35] DESCRIBE provides an environment that supports reference by description in communication between human and machine. We find it helpful to view this environment in light of the natural environment that supports reference by description in communication between humans. In natural language, the primary concern is the description of *events*, which in turn requires the description of the *objects* participating in these events. Practicality demands that descriptions be of manageable length; in interpersonal communication, this is achieved by both speaker and listener taking into account the relevance of the objects and events in the immediate context.

Table 5 indicates the relations between these natural language concepts and those present in DESCRIBE. For comparison, approximately corresponding UNIX concepts are also indicated. Each of these correspondences will be considered in turn.

Object description in natural language corresponds to file description in DESCRIBE. In DESCRIBE there are several kinds of *file descriptors*: *types*, *properties*, and *relations*. These are the analogs of the object descriptors that appear in natural language: common nouns, adjectives, and relational predicates, respectively.

Common nouns classify objects and play a crucial role in object description. In DESCRIBE, a similarly central role in file description is played by *file type* descriptors such as *text*, *lisp source*, *message*. Like common nouns, file types form a rich classification hierarchy. [36]

The objects in a given noun category can be described by certain associated adjectives. An adjective, such as *enormous*, can be thought of as a value, *extremely positive*, for some property, *size*. In DESCRIBE, a single file can be described by specifying *values* for a number of *properties* like *creator*, *project*, *protection*. Values can be specified at a variety of levels of detail; the possible values for each property form a hierarchy. The properties applicable to a file are determined by its type. For example, the properties specifically applicable to files of type *message*, include *recipients*, *header*, and *reply to*. [37]

Describing an object with a relational predicate amounts to specifying a relationship holding between the object being described and some other object(s). In DESCRIBE, files can be described by specifying relationships to other files through *relations* such as *revision of*, *response to*, or *compilation of*.

---

35. It is important to remark that many of the capabilities of DESCRIBE can be achieved - and in many cases already are achieved - in other ways. Our claim is that our analysis provides a *natural*, *coherent* framework that can lead to enhanced human/machine performance. The approach can be viewed as part of the investigation by the UCSD HMI Project of a general and powerful hypothesis: human/machine performance can be augmented by making available *within the machine* some of the *meta*-information that now only users possess about the objects and activities they create in the machine. The UCSD Project uses the UNIX operating system as its point of departure.

36. In a DESCRIBE "hierarchy," nodes may have multiple parents (e.g., a project *IFIP abstract* that is a subproject of two other projects *IFIP paper* and *progress report*). A more precise term would be "directed graph."

37. Files of a given type inherit the applicability of properties from their ancestors in the file hierarchy.

Table 5

Interpersonal Communication	Human/Machine Communication	
	DESCRIBE	UNIX
Object description (multiple hierarchies)	File description (multiple hierarchies)	File naming (single hierarchy)
Event description	Process description	Command line
Context	Workspaces	Current working directory

Descriptors allow users to describe files rather than name them. [38] Descriptors also serve to *organize* files: the hierarchy of values for a given property induces a hierarchical organization of all files from the perspective of that property. By focusing on various perspectives, users can dynamically choose from among the many such organizations those best suited to the present need. In this sense, *each property* corresponds to the hierarchical file organization (directory structure) of UNIX. As the example of the second paragraph shows, the independent classifications offered by the properties of DESCRIBE must be jumbled together haphazardly in typical UNIX hierarchies because only one such hierarchy is available.

In DESCRIBE, description forms the basis not only of file retrieval but also of command specification. Specifying a command is viewed as *requesting the operating system to create a process matching a given description* (see O'Malley et al., 1983; Smolensky, Monty, & Conway, 1984). Describing a process corresponds in natural language to describing an event. Such descriptions entail classifying the event with a verb, possibly qualifying the event with an adverb, and specifying for each role in the generic event the object that fills that role in the particular event.

An event is approximately classified by a verb; a process is precisely classified by the executable program it runs. This classification forms a natural extension, from files to processes, of the concept *type*. The "options" in command specifications correspond to adverbs, and form a natural extension of the concept of *property* from files to processes. Finally, the files that serve as arguments to commands correspond to the objects filling roles in generic events. The links between these argument files and the process being created comprise a natural extension of the concept of *relation* to include those that involve processes as well as files. Thus, for example, the UNIX command line *cc -O foo.c -o run* becomes in DESCRIBE a description of a process of type *compilation* with: property *optimize?* set to *true*, relation *source code* assigned to the file named *foo*, and relation *executable program* set to a file named *run*. [39]

Thus the same framework that was needed to support file description suffices for process description, i.e., command specification. To summarize: DESCRIBE permits users to describe both types of *structures* – files and processes – using relations that associate structures with other structures, and properties that associate structures with values. Furthermore, structures have types that determine which properties are applicable to them.

Descriptions must be kept manageably short if they are to be useful. One source of economy in DESCRIBE takes advantage of the selection restrictions that hold between processes and files. Whenever a file is being described as part of a command, the range of possible file types is delimited by the role (i.e., relation) that the file fulfills. (So if several files named *foo* exist but only one is of type *source code*, *foo* alone suffices to describe the file satisfying the relation *source code* for a *compilation* process.) [40]

---

38. *Name* is just one of many properties for which files have values.

39. The files could of course be described rather than named.

40. Two important corollaries of this observation further indicate the value of typing structures. If the only file matching a user's description has the wrong type to satisfy a process relation, a meaningful error message can be offered and the often undesirable results of running a program on an inappropriate file avoided. Secondly, semantically closely related operations (such as pretty-printing Lisp code and pretty-printing C code) that must be implemented differently for different file types can be fused into a single command, with different implementations invoked depending on the type of the given file.

Descriptions are kept manageable in natural circumstances largely by the limitations imposed by relevance. In DESCRIBE, the set of structures relevant to a particular working environment are brought together by a user into a *workspace*. Workspaces are defined primarily as hierarchically organized collections of functionally related files and processes, and are used to help organize the activities and goals of the user (see Bannon, Cypher, Greenspan, & Monty, 1983). Within DESCRIBE, workspaces are constructed through a readily user-expandable set of values for the relation *member of*. Descriptions that would be ambiguous in the absence of context are disambiguated by finding a plausible referent in the *current workspace*. This function is performed in UNIX by the concept of *current working directory*. [41]

The well-defined characteristics of a workspace can be used to automatically assign default values to some properties of the structures created within the workspace; in effect, each workspace is a personalized computing environment tailored to support work on a single task. This, together with the significant number of properties that are determined by those processes that create structures, controls the burden on the user of describing structures as they are created. It is also important that file descriptions can be added long after creation time, as the needs of file organization grow and change.

In conclusion, reference by description offers an attractive alternative to reference by name as a basis for operating system design: it provides much of the power and flexibility found in interpersonal communication. We emphasize that despite the pervasiveness of natural language analyses throughout this paper, we are *not* proposing any form of "natural language interface." Rather, we are suggesting that the *abstract structures* underlying communication and reference in natural discourse offer a sound foundation for communication between human and machine.

References

Bannon, L., Cypher, A., Greenspan, S., & Monty, M. L. (1983). Evaluation and analysis of user's activity organization. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human Factors in Computer Systems*. New York: ACM.

Norman, D. A. (1983). Design principles for human-computer interfaces. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human Factors in Computer Systems*. New York: ACM.

O'Malley, C., Smolensky, P., Bannon, L., Graham, J., Sokolov, J., & Monty, M. L. (1983). A proposal for user centered system documentation. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human Factors in Computer Systems*. New York: ACM.

Smolensky, P., Monty, M. L., & Conway, E. (1984). *Formalizing task descriptions*. Manuscript submitted to the IFIP Conference on Human-Computer Interaction. (London, September 1984). Also included in this technical report.

---

41. In UNIX, the grouping of files for descriptive purposes and for working purposes are confounded within a single directory hierarchy. In DESCRIBE, these functions are independent.

CAVEATS ON THE USE OF EXPERT SYSTEMS

Liam J. Bannon

Recently we have witnessed a round of assertions and counter-assertions about the capabilities of applied Artificial Intelligence, specifically in the area called "knowledge engineering," where scientists are involved in the building of so-called "expert systems" that are designed to mimic the performance of human experts in certain domains. Strong claims about the potential social benefits of such systems are being voiced by people within the AI community, but what is especially interesting is that the business world has decided to invest in these AI developments. The questions I wish to pose concern the potential social ramifications attendant on the widespread use of these expert systems.

What are expert systems? In brief, expert systems consist of a "knowledge base" which consists of large numbers of domain specific rules together with some form of "inference engine" which can draw inferences from the corpus of rules in the database. Current technical debate focuses on such issues as the wisdom of a clear separation between data and inference procedures and the relative strengths of different knowledge formalisms. One problem in the area comes from the fact that much of the reasoning of human experts is done under conditions of uncertainty, which therefore rules out the use of such powerful inferencing systems as the predicate calculus. Another stems from the size of the solution space—even in quite narrow task domains it is liable to be vast, ruling out simple search methods such as exhaustive search. Expert systems must therefore incorporate heuristics to reduce the search time and make the problems solvable. (See Davis, 1982, and Stefik et al., 1982, for a technical introduction to the area.)

Typically, the knowledge or "beliefs" of the expert system are built up painstakingly through interactions between AI researchers and experts, involving numerous iterations with the evolving system, adding new knowledge to the system until its answers appear to model those of the experts in some consistent fashion. Over the course of time, it appears to be possible to build a system that has quite impressive deductive powers in a limited domain. (See, for example, the DENDRAL (Lindsay et al., 1981) and R1 (McDermott, 1981) systems.)

My concern is that rather than being seen as legitimate research tools that might further our understanding of knowledge representations and the nature of human expertise, these expert systems may be viewed from a narrow economic perspective as simply reducing the need for highly-trained specialists. This could lead to problems on several levels. The nature of the human-machine relationship could be adversely affected, as the less-skilled operator of the intelligent system might feel unable to query the findings of the system, and unable to understand the reasoning behind the system's decisions, even if the system could provide some explanatory capability. It is also possible that the system might give information, or suggest courses of action, that are unsuitable to the client's specific needs due to a misperception of the original problem. Of course, human experts are fallible also, but they do bring a variety of talents to bear on a problem that are as yet untouched by expert systems.

As Flores and Winograd (in preparation) have noted, the label "expert system" has misleading connotations, as expert systems are in about the same relationship to real human experts as are idiot savants. They state that one does not refer to an "idiot savant" as an expert, precisely because the capabilities of the idiot savant are so limited to a particular domain, showing no generalization, and inflexibility. Human experts differ from expert systems, not only in being able to "go beyond" their rules and restructure their knowledge at certain crucial points, but also in being able to reflect on their knowledge, and they are always located in a social context which influences their decisions. This sensitivity to the social and cultural context of the situation is especially crucial in situations involving medical diagnosis and treatment.

This is not to say that "expert" systems are useless, only that their successful use will be confined to narrowly defined domains, where there are a limited number of objects in the task domain, and a well-specified set of relations between the objects. The designer of the expert system has to explicitly encode these relations into the system, and the system will always be constrained by the omissions of the designer. Of course, new "knowledge" can be added to the system when a problem occurs, but only if someone explicitly changes the system. This points out a fundamental limitation of these systems; they cannot, at least to date, learn from their experience, a prerequisite for any truly "intelligent" system.

If we bear these points in mind, and reflect on the use of expert systems in such sensitive domains as medical diagnosis and treatment, one can see why there are reservations in allowing an expert system to operate in an essentially autonomous fashion. It is impossible for any system to be able to take into account the full context of the situation, as in any person-to-person encounter there are a myriad of potential signals that, on any one occasion, might be important in diagnosis or treatment of the patient. What is and what is not relevant is extremely difficult, if not impossible, to determine in advance, and thus any technical system will be bounded by the inventiveness of its creator, no matter how insightful that person is. This is not to argue against the use of computer systems in the diagnostic process but to assert that it should be under the control of a fully trained physician who feels comfortable in "going beyond" the system on occasions. The specter of poorly trained medics in supplication to "THE EXPERT SYSTEM" which they have been told holds all relevant medical knowledge strikes me as fundamentally unsettling, both from the point of view of the medics themselves, in their feelings of loss of control over the situation, and of the patients, in their feelings of uncertainty about the treatment suggested. It brings to mind the scenarios explored by writers such as E. M. Forster in his short story *The Machine Stops*, of an unbounded autonomous technology which has gone beyond human control and comprehensibility.

My argument is against certain "non-expert" uses of expert systems, but unfortunately, I believe that it is these kinds of uses that are of potential interest from a commercial standpoint, as expert systems are seen as one way to reduce the high costs involved in the utilization of human experts. I am concerned that potential users will ignore the risks involved in the general use of fully automated diagnosis and treatment systems. When serious mistakes occur, as they inevitably will, one can see the technicians saying, as those at Nuremberg did: "I was only doing my job; this is what I was told to do (by the State, or the expert system)." (See Weizenbaum, 1976, for an elaboration of the potential misuse of complex computing systems.) There comes a time when people do not feel in a position to override the system, or else shield themselves from decision-making responsibility under the guise of machine *dictat*.

In summary, this is not an outright attack on technology, or even AI, but a cautionary note against an uncritical view of the social benefits to be gained by automating the capabilities of human experts in every field of human endeavor.

References

Davis, R. (1982). Expert systems: Where are we? And where do we go from here? *AI Magazine, 3,* (2), 3-22.

Flores, F., & Winograd, T. (in preparation). *Understanding computers and cognition.*

Forster, E. M. (1928). The machine stops. In *The eternal moment and other stories.* Harcourt, Brace and World.

Lindsay, R., Buchanan, B. G., Feigenbaum, E. A., & Lederberg, J. (1981). *Applications of AI for organic chemistry: The DENDRAL project.* New York: McGraw-Hill.

McDermott, J. (1981). R1: The formative years. *AI Magazine, 2,* (2), 21-29.

Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., & Sacerdoti, E. (1982). The organization of expert systems: A tutorial. *Artificial Intelligence, 18,* 135-173.

Weizenbaum, J. (1976). *Computer power and human reason.* San Francisco: W. H. Freeman.

SOFTWARE ENGINEERING FOR USER INTERFACES

Stephen W. Draper and Donald A. Norman

The discipline of Software Engineering can be extended in a natural way to deal with the issues raised by a systematic approach to the design of human-machine interfaces. Two main points are made: that the user should be treated as part of the system being designed, and that projects should be organized to take account of the current (small) state of a priori knowledge about how to design interfaces.

Because the principles of good user-interface design are not yet well specified (and not yet known), interfaces should be developed through an iterative process. This means that it is essential to develop tools for evaluation and debugging of the interface, much the same way as tools have been developed for the evaluation and debugging of program code. We need to develop methods of detecting bugs in the interface and of diagnosing their cause. The tools for testing interfaces should include measures of interface performance, acceptance tests, and benchmarks. Developing useful measures is a non-trivial task, but a start can and should be made. [42]

Introduction

The subject of this paper is the extension of Software Engineering to deal with the issues raised by the design of human-machine interfaces. To a large extent, all that is needed is to take the problem of engineering the user interface as seriously as any other part of software engineering and to apply to it the same kind of techniques, appropriately adapted. For instance, although the interface is implemented in software, it can be thought of as being "run" on human users. This means that we must modify our concept of a program *bug* to allow for part of the system to be a person; we must establish new performance criteria for the combined human-plus-interface system. Much of the thrust of this paper is simply to draw analogies with existing software practices in order to suggest how to support a professional approach to interface design. We do not present detailed ideas on what interfaces should be like, [43] but rather sketch some consequences for software engineering when interface design is taken seriously.

There are two themes in this paper: arguing by analogy with existing practices to their extension to interface design; and arguing from the nature of the problems of interface design to requirements for an appropriate engineering discipline. The first part of the paper makes some general points, the second summarizes their consequences for the coding, documentation, debugging, and testing phases.

---

42. Published in the *Proceedings of the Seventh International Conference on Software Engineering*. (Orlando, Florida, March 1984). We thank Tony Wasserman, Eileen Conway, and Sondra Buffett for their assistance with the manuscript.

43. We leave out of this paper any major discussion of what a human-computer interface ought to be, or what the general problems or principles are, because this topic is explored in numerous other papers by us and by the growing community of people who work in the field called *Human-Computer Interaction*. This field now has an annual meeting, several journals, and an ACM Special Interest Group (SIGCHI): we see no need to cover the material here. See, for example, the *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems*. This paper concentrates on the lessons that can be applied to interface design from practices already common in Software Engineering.

Software Engineering for Human-Computer Interface Design

Two Goals in Optimizing an Interface

We begin with a tradeoff that has clear parallels to aspects of software engineering, the tradeoff between two quite different broad aims for an interface:

- Achieving speed and convenience of use (*power*) for the practiced user;
- Achieving ease of learning and use (*ELU*).

These distinctions are related to the familiar novice-expert distinction. There is a clear analogy between these aims and the desire to optimize space and time in software performance. In general it is desirable to optimize both aims, but beyond a certain point, the engineer must choose a particular tradeoff between the two. This analogy holds quite closely. Not only is there a reasonable region where one aim trades off against the other, but, at the extremes, the programs can be generally unusable.

Consider the extremes. Suppose we optimized program speed or interface power. Just as a fast program that requires more space than any customer can afford is in practice useless, so too is an interface that provides immensely fast and "powerful" interaction, but at the cost of requiring such a level of skill that no user will actually be able to use it because of the difficulty of learning it presents. Similarly, if we optimized program space or ELU, the program might be too slow to use regularly (no matter how economical of space) and the interface can be too laborious for a regular user to employ productively, no matter how safe, "friendly," helpful, and easy to learn. If we avoid the extremes of these dimensions, then there is a wide range of choice for the designer. The existence of programs of similar function on microprocessors and large mainframe computers is a strong reminder that we can tradeoff both in the space-time domain for the software and in the power/ELU domain for the user interface.

User and Program as Communicating Co-Routines

When a programmer implements a system with a user interface, he or she is not only defining what the machine will do but also defining what the user can do. The program and user can be thought of as co-routines, each communicating with one another. The programmer must explicitly or implicitly decide what actions and information will be available to the user. For any given interface, this point can be illustrated by drawing a flow chart of the user's part of the process. Although this point seems simple, it changes the fundamental (though usually unacknowledged) situation by which programming takes place. No longer are programmers engaged in the private task of getting the machine to do something for themselves — they are not even engaged in communicating a program to other programmers, a view intermittently voiced by purist programmers. Instead, the interface designer should be viewed as someone who must write a successful co-routine between user and machine, yet where the full specification of the user side of the co-routine is not well known (and may be variable): this is the real subject of the design, and this must become the conscious objective.

There are two consequences to this point, both discussed in greater detail later in the paper. First, we need languages that support this view, that is, that represent this co-routine directly. Second, the interaction needs testing and debugging on typical "hardware," hardware

that includes representative users.

The State of User Interface Design

How much is known about how to design interfaces? We must answer this question in order to plan a sensible software engineering strategy. If enough is known to lay down detailed principles, then a top-down strategy might be followed; otherwise an iterative strategy must be adopted with emphasis on the testing and debugging phases.

Quantitative principles. Top-down principles of design should, in the ideal case, allow a design to be worked out in advance rather than entirely by trial and error. These tools need to be developed from a solid basis in experimental psychology, coupled with a good understanding of programming and software design. Not surprisingly, the number of groups capable of this work is limited and not much yet exists. There are now several initiatives whose goal is to provide quantitative principles for the design of human-computer interfaces. For example, one of us has shown how it is possible to develop a quantitative assessment of how design tradeoffs affect the *User Satisfaction* for an interface (Norman, 1983). Thus, in trading of time, information, and workspace, it is possible to compute the tradeoff space, showing the psychological impact of tradeoffs in these variables. A complementary approach is that of Card, Moran, and Newell (1983) who provide a set of quantitative tools for computing the operational parameters of interfaces.

The development of psychologically based, quantitative design tools is still in its infancy and much work remains to be done, both in development and in validation. If we can develop sufficient range and breadth of quantitative design tools, then we can look forward to a time when it will be possible to provide general principles and even tables of numerical values in design handbooks, allowing such design considerations as workspace size, display time, menu structure, command language design, pointing-versus-naming, interface power, and ELU to be assessed at the design stage, without the need to build a test system. For the present, however, the designer, by and large, does not have quantitative data at hand.

Qualitative principles. In the absence of well-established quantitative design aids, we need methods that allow us to work with qualitative principles. A large number of qualitative principles exist, many in the form of "slogans," exhorting the designer to consider this factor or that, or to avoid this failing or that (see for example, Badre & Shneiderman, 1982; Nievergelt, 1982a, 1982b; Shneiderman, 1980). These qualitative rules are often quite reasonable (which makes it especially discouraging to see how frequently even the most obvious and elementary principles are violated in existing systems), although if the design of a user interface is ever to proceed in a systematic fashion, we need to go beyond this level.

One major example of the attempt to base a system design on fundamental design principles, built in such a way as to capitalize on the user's existing knowledge is the design of the Xerox *Star* system (Smith, Irby, Kimball, & Verplank, 1982). This design attempted a systematic strategy based on principles of good human-computer interaction. Since then, of course, other systems have been developed along similar principles (in particular, the Apple *Lisa* system). The *Star* illustrates the delicate tradeoff decisions that must be faced: the attempt to optimize ELU led to degradation of performance, especially in speed of the system, as well as to a high selling-price.

Specifications for Interfaces

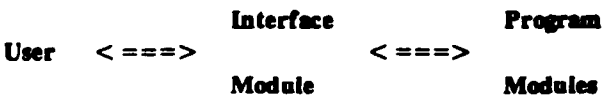
In defining and addressing a software project two questions must be tackled: what kind of specifications are reasonable for defining the project?, what kind of divisions of the project into subtasks are likely to be viable? The same observation applies to both: specifications are determined at an early stage and must remain constant if massive re-writing is to be avoided.

Sheil (1983) argues against the utility of a Structured Programming approach to user interface design. Sheil points out that whatever the virtues of that approach, it depends on having a clear and fixed specification at the start. The technique becomes strained whenever the specifications change during the life of the software. Although in the ideal case a Structured Program practitioner would re-do the whole design and implementation at every change in specification, in practice the investment in the existing code and design exerts an enormous pressure toward piecemeal change, i.e., toward iterative redesign. In areas where such shifts in specification are common, it seems better to face the fact that development will in practice be iterative. Sheil suggests — and we agree — that interface design is one such area.

Several consequences follow from this point of view. First, it will be unwise to allow any details of a user interface to appear in the project specifications: for instance instead of giving a list of the commands to be implemented, specifications should rather be of the form discussed by Shneiderman (1982), who suggests instituting acceptance tests for interfaces such as "after 75 min. of training 40 typical users should be able to accomplish 80% of the benchmark tasks in 35 min. with fewer than 12 errors." A professional approach to user interface design puts the responsibility with the designer, having the user or customer specify performance requirements rather than details, just as in Shneiderman's hypothetical acceptance test. This leaves the design of the interface to the software team, but with explicit standards for the specification of the performance of the system.

A second consequence is that we must expect to have to go through many iterations on details of the interface. If the overall project is to be subdivided by dividing the system into modules that are separately designed and implemented, then a third consequence is that the user interface should be segregated into a single module which all other modules must use if they communicate with the end user.

Separating interface design from program design. This strategy consists of isolating the main program in one set of modules, the user interface in another, and considering the user as comprising a third:



The virtue of this view is well known in the software engineering community: as long as the communication structure and the data representation are well specified and adhered to, the modules can be worked on independently and changed at will, without any effect on the rest of the system. The communication between the *Interface* and the *User* can be changed in any arbitrary manner as long as the communication between the *Program* and the *Interface* is not affected. This requires, of course, that the only part of the system that may interact directly with the user is the *Interface Module*. A well-defined protocol between the main program and

the interface module can be defined at the outset, leaving the interface designer free to change the interface without interfering with the independent development of the main program. An important benefit of this modularization strategy is that it allows separate specialists to work on the main program and on the interface module.

A good example of this modularization combined with support for a heavily iterative approach to developing the user interface is provided by the database and interface packages *Troll/USE* (the database: Wasserman, van de Riet, & Kersten, 1982) and *RAPID/USE* (the interface package: Wasserman & Shewmake, 1982). *Troll/USE* is a relational database — the "main program" module. *RAPID/USE* is a tool for the design of interactive dialogues and systems, allowing for rapid modification and experimentation of the interface (through a transition network specification), yet maintaining constant the necessary communication protocol to the database, *Troll/USE*. Clearly the design of a database requires different skills than designing a user dialogue, and in fact *RAPID/USE* is designed to allow non-programmers to design the dialogue using a simple interpreted language. Other examples are the DMS system (Roach et al. 1982) and the work by Buxton et al. (1983).

Makeup of interface design teams. A further consequence of the special nature of interface design affects the organization of the design team. If the principles of interface design were developed to the point where their application required no personal expertise, and if the tradeoffs were all identified and quantified in advance, then a split between groups of experts might be workable. It is noticeable that some of the more successful interface designs (e.g., the Xerox *Star* and the Apple *Lisa*) have been done by teams which included people with diverse talents and training, but which did not make a distinction between evaluators and implementors, between psychologists and programmers. We interpret this to indicate that at the present time the closest cooperation is required to identify and resolve the unexpected tradeoffs that surface in the course of a design. Many design questions are at heart unforeseen tradeoff decisions, and these can only be reasonably made by people who appreciate all the factors involved. This, although a separation of the interface from other program modules is highly beneficial, and although logically the skills required for designing each are quite different (e.g., dialogue design versus application programming), it seems that the unknowns in interface design continue to demand close cooperation between parts of design teams.

Influences of the Interface on Code

Languages for User Interface Design

We have argued that interface designers need to be aware that they are designing a dialogue, and more than that, a co-routine between user and computer. The idea immediately follows that special languages that represent this view would be an important aid. Certainly conventional languages are poor in this respect because they are machine-centered: they describe events around the sequence of machine actions with input and output seen as side-effects. Wasserman's *RAPID/USE* system offers the designer a language based on Transition Diagrams, where nodes correspond to a machine state and the output displayed at that point, and the various arcs from a node correspond to alternative user inputs. This is a major step in the right direction. It is perhaps not a complete solution because it is essentially a representation of the machine the user sees (as a quasi-finite state machine) rather than of the co-routine. The user's processes are only implicit (though much easier to see than in ordinary languages), and machine actions are represented only as side-effects of transitions. Until further advances in this area

are made, exercises such as flow-charting the user's decision-making and actions will remain important.

The Interface to the User Module

We have argued that the user should be viewed as a module of the system. Good programming practices now demand that one establish a uniform communication protocol early in the design phase and stick to it. Part of the rationale is economy — if module U has to use one protocol for interacting with module A and another for module B, it will take more code to do so. (Or, by analogy, if you think of module U as being the User, then the user has more to learn.) In addition, there is scope for confusion and error in the programming if there is not a uniform protocol (in the analogy, the lack of consistency makes it harder for the user to learn and to apply the protocols).

These arguments apply best to low-level user protocols. The more programs use a given protocol for interaction, the more benefit the user gains from the associated set of skills in using it (i.e., from their associated "subroutines"). Thus a user is helped if all parts of the system use a small number of common protocols for interaction, protocols that the designer can think of as corresponding to subroutines in the user's mind. [44]

It is not sufficient to think of the user as a simple system module with a single protocol for interacting with other modules. There are many layers of protocol, just as there are in computer networking. At least three levels may be distinguished: (1) the low-level protocols just discussed; (2) the conceptual model of the domain the designer wishes to present to the user; and (3) the highest level where the user has several concurrent goals (e.g., to send a letter or get a budget analysis) and the computer is one means to the ends. Most of our knowledge about human-computer interaction is at the first two levels: little is understood/known about the highest level of user goals. The Xerox *Star*, the Apple *Lisa*, and *Visicalc* can be seen as paradigm examples of the second level in that the system image was decided early on as a major design decision which defined the context in which the rest of the interface was developed.

Standardized Packages

A major development tool would be the creation of various packages for doing standard interface operations. There are two separate motivations for this:

- To provide a language at the right level for the designer (where the operations are elements of user interaction and lower-level details can be hidden);
- To provide standard modes of interaction across the system to help the user.

The first provision helps the designer to work on the design of the interface undistracted by implementation details. The second provision gives a standardization (consistency) across applications that helps the user, as argued above. Simply providing packages is often enough to get standardization — it makes it easier for a programmer to conform than to dissent. There is

---

44. A major problem here is to ensure a match between the actions expected of the user and the capabilities of that user. This is one of the major themes of research on human-computer interaction, and is a non-trivial problem. In general, we would submit that here is where the software designer must interact with a human-factors or psychology software designer — during the design phase — to develop the specifications of the user actions.

great need for software tools for interfaces, including screen management, user-program dialogue, and packages for doing help, argument parsing, history, and undo. A good example is Perlman's (1983) general interface package.

Documentation as an Integral Part of the Interface

An obvious consequence of integrating user interface design into the overall software engineering is to integrate documentation and code generation. Mashey and his colleagues at Bell Labs have taken a major step in this direction by using the same file control system for both source code and documentation (Bianchi, Glushko, & Mashey, 1982), thus promoting their simultaneous development and allowing immediate checks on whether one is out of date relative to the other. Knuth's recent work is along the same lines (Knuth, 1983). This is most likely to benefit other programmers (rather than users) who will have to maintain the code, since they are often the major beneficiaries of complete and correct documentation. However, if it is true that no interface design is likely to remain fixed for long, then it is not enough for it to be friendly to the user — it must be friendly to its maintainers as well.

The term "documentation" should not be viewed too narrowly. Users get information from a number of sources including manuals, tutorials, error messages, and normal displays. One test of the adequacy of the overall documentation is to introduce an error in the operation of the system while observing a "typical" user in a "typical" situation. The observation of interest is to determine how the user copes with the situation: the design fails the test if the user cannot recover gracefully. It is important to note how the user determines the state of the system and the options that are available, and also to observe what the user actually does (which may be quite different from what the designer had in mind).

The success of the system will usually depend on a combination of information sources and is neither a property of the code alone nor of the documentation alone. The theme is that in order to provide good documentation from the user's point of view it is necessary to identify what information the user needs, and when. Then, it is necessary to provide a channel from the situation to the information. It is relatively unimportant what media are used for this in any given case: they could be messages generated by the system or notices stuck to the terminal. What matters is whether the user is able to get the information that is required. Note that it is not relevant that the information is available in principle. What matters is whether real users, in real situations, can get the answers. If the user cannot solve the problem, then it is the system design that is at fault, whether the designer can demonstrate that the relevant information was available to the user: the critical test is the practical one — do real users succeed at the task?

Debugging the Interface

When a piece of software has been implemented it needs to be tested and debugged: the same applies to the user interface. In the past, debugging the interface has generally been left to the customer, whose complaints are classified as changes to the specifications. The net effect tends to be that changes are slow, expensive, resented by the programming team, and do not benefit from any kind of systematic or professional approach. Clearly the field is more than ready for improved practices.

What Is a Bug?

The field of debugging involves many issues. One problem is to determine what counts as a bug, another is to determine what symptoms can be detected in practice (and what proportion of bugs escape because they produce no clear symptoms), and yet another to determine the cause from the symptom. The concept of "bug" is clearly useful in both traditional and interface software engineering, but nevertheless it has no clear definition. Some bugs are clear — if an explicit specification is not met, the implementation has a bug. However there can be bugs in the specifications themselves, and bugs relative to implicit specifications. A crucial part of developing interface engineering will be developing standards that become implicit specifications for all interface programming. (The analogous points for bugs in programs are discussed in Johnson, Draper, & Soloway, 1983.)

We believe that the system specifications should include a statement about the class of user and the kind of training that is to be expected. The system should then be evaluated with that very same class of user, with the same training procedure. If the user then has problems, there is a bug in the system. The bug could be in the training, or in the interface. The point is that we cannot determine just where the problem lies until we have explicit specifications for all aspects of the computer system, including the interface and the user performance. When we have specifications that cover the user, then we can determine how reasonable they are on the basis of the user's abilities. Only when we have determined that the specifications are indeed reasonable can we then claim that the system that fails to meet those specifications is at fault. This lesson applies to all parts of the system, of course, but its implications for assessing the role of the user as a part of the overall system operation seem not to be properly recognized.

Finding Bugs

The only way to find bugs is to test. This means that the system must be put through its paces with the human user, much as programs are put through their paces with test sets of data. Just as a program needs testing by data that exercise every branch of the code, so the user-program interaction needs testing by exercising each possible "branch" of the interaction. Unfortunately, test procedures for user interfaces do not exist.

Note that the testing phase is not apt to be easy. It requires the development of good test problems, of a good pool of users upon whom the tests will be run, and careful observation and evaluation of the result. It is critical that the users upon whom the system be tested reflect the actual user population for whom the system is designed. Psychology has amassed a number of methodological tools that can be of use. Other tools, specialized for this particular problem, need to be developed.

Learning how to ask users for information is as big a topic as learning how to extract useful measurements from computers. For instance, consider a faulty error message. If it is so useless that the user cannot understand it and gets stuck, there is often a bias against reporting the consequent failure to carry out the task successfully because the users are apt to feel that the problems are due to their own inadequacies. On the other hand, if the message is wrong

or silly in some way but nevertheless the users succeed in diagnosing the real problem fairly quickly, then they are likely to express their irritation. Note that this means that the non-fatal inadequacy is likely to receive a much higher rate of spontaneous complaints than the much more serious case which causes users to fail completely. Obviously we need to learn how to work around phenomena like this. For instance, using exhaustive checklists in questionnaires (Root & Draper, 1983) ensures that one solicits opinions on all parts of an interface, and, to some extent, allows one to see things such as mass avoidance of a command that no-one complains about spontaneously .

People are very sensitive to the context in which they are operating, and if one is not careful, the test population may feel that it is they who are being evaluated (rather than the system), and they may carefully monitor their responses and behavior so that they will not "look bad" or "stupid" (Lewis, 1983). One of us experienced the situation where a deficiency in the system was not reported by any of the users because they attributed the difficulty to their own inadequacies, not realizing that it could be avoided by a (rather simple) design change. In this case, it required an experienced observer to watch users and note the problem. Note also that the existence of any problem was at first denied by the design team who asked "but why has nobody ever complained?" This sounds reasonable, but is analogous to a programmer who does no systematic tests and then uses the length of time before the first complaint as evidence that complaints must be ill-founded. This is not a trivial instance: users who feel that a system reveals their inadequacies will not wish to use the system and will resist its introduction into the workplace. Thus, the system will not get used (or morale may suffer). The problem is to devise techniques that allow the designer to realize the nature of the difficulty. It will take extreme sensitivity on the part of the tester to overcome these problems. It is here that the skills of the experimental psychologist are probably essential.

Debugging Tools

The use of questionnaires is analogous to a post-mortem in that they are applied after the program has run. One of the most pressing needs for interface debugging is to have facilities analogous to run-time tests built into all computing environments that cause program exceptions for bad addresses, floating point overflow, etc. Although an important function for these is, of course, the protection of other users, they are also valuable for debugging because they stop execution at the earliest sign of trouble and give the programmer a chance to gather information on the state of the process at that point. Ease of debugging is crucially affected by the immediacy of error detection, as anyone knows who has debugged programs with and without array bounds checking. Applied to interface debugging, this means developing suitable error criteria, and then acting on it. It is not necessarily appropriate to halt a program when an error in interface interaction is detected, but at the least, one could create a relevant "dump" — a trace of the whole interaction together with as much information as possible on the users, their experience, and their current goals and thoughts at the time of the difficulty.

The various existing techniques for getting at the interaction between the user and the system differ in their immediacy and the information provided. Furthest removed from the actual interaction is the collection of opinions after some amount of experience with the system (e.g., at completion of the training period). Closer to the actual usage is the use of on-line complaint facilities. A still more immediate record is provided by history traces or dribble files

that provide a detailed record of the low-level actions of the user, but without any of the goals or intentions. Intentions and goals can be gotten by the collection of real-time, thinking-aloud protocols from users while they interact with the system. Each technique offers a different perspective on the interaction.

Testing the Interface

Improving Measures of Performance

In addition to debugging, a programmer will typically be concerned with examining and optimizing certain measures. The best parallel here is with the problem of improving a program's speed of execution. The conventional wisdom on the timing problem is that a typical program spends 90% of its time in 10% of its code, so the strategy is to identify that 10% and work on tuning it, since work on improving the other 90% will show little effect overall. Thus, profiling tools that show where a program spends its time are important. In improving an interface, several issues are relevant: how many users find a given command problematic?; how problematic do they find it?; how often does the issue arise? As with debugging, we see here a gap between what can be easily and directly measured and the underlying concern of the designer.

Like profiling tools, then, interface tools should produce measures of those things that can be used by the designer to pick the next point of attack, together with a measure of how important it is to do any further improvements. Also like profiling tools, there will be issues of how accurate these measurements are (resolution difficulties) and how representative of the real situation. Ultimately a lot will also depend on the experience and judgment of the designer using the tool. Thus not only do the tools need to be developed, but it will then take a further significant amount of time to accumulate experience in the use of these measures.

Another tool is on-line command usage measurements. It is relatively easy to collect a running record of command use for the various users of a system, thus providing reliable measures of how often commands are used, and by what percentage of users. The frequency of use is important in weighing the priority to be given to problems. [45]

Benchmarks and Acceptance Tests

Earlier, we discussed Schneiderman's (1982) suggestion that the specification of the interface be given in terms of the acceptance tests to which it will be subjected. This idea can have far-reaching effect in focusing designers' attention on a definite goal for the interface. Whether success at a particular test turns out to be a good guide to the user's long-term satisfaction with the product, it is at least at an appropriate level of specification; this is a crucial step in extending software engineering to interface design.

---

45. There is a major problem of invasion of privacy. It is not appropriate to keep records on the details of individual users' interactions with a system. Our solution is to encode the user's identity so that although the user identity cannot be determined, we can still match up the particular command sequences and program usages with the user codes. This is essential in allowing the discovery of common patterns of operation.

User-interface benchmarks will be most clearly useful when the aspects of performance and the situation in which it is to be measured are clearly defined. As a general method by which to judge a whole system, benchmarks are obviously limited; systems differ on many dimensions and benchmarks often generate only a single measure. The use of benchmarks for interfaces is further problematic in these early days since we do not yet know all the crucial variables. For instance, discussions about which of two operating systems are more effective for a class of users are sometimes carried on without considering the communication rate of the channel to the user, yet this crucially affects how much feedback is perceived as a painfully time-wasting nuisance. In general, factors not directly under the control of the engineer may have a dominating effect. Until we are more confident of being conscious of the factors that have a major influence on the measures we are interested in, we will not know how to run benchmarks in which they are held constant. [46]

References

Bianchi, M., Glushko, R., & Mashey, J. (1982). A software/documentation development environment built from the UNIX toolkit. In H. J. Schneider & A. I. Wasserman (Eds.), *Automated tools for information systems design*, (pp. 107-108). Amsterdam.

Badre, A., & Shneiderman, B. (Eds.). (1982). *Directions in human-computer interaction*. Norwood, NJ: Ablex.

Buxton, W., Lamb, M. R., Sherman, D., & Smith, K. C. (1983). Towards a comprehensive user interface management system. *Computer Graphics*, 35-41.

Card, S., Moran, T., & Newell, A. (1983). *The psychology of human-computer interaction*. Hillsdale, NJ: Erlbaum.

Janda, A. (Ed.). (1983). *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems*, New York: ACM.

Johnson, W. L., Draper, S. W., & Soloway, E. (1983). Classifying bugs is a tricky business. *Proceedings of the Seventh Annual NASA/Goddard Software Engineering Conference*. Baltimore.

Kersten, M. L., Wasserman, A. I., & van de Riet, R. P. (1982). *Troll/USE reference manual*. San Francisco: University of California, San Francisco, Laboratory of Medical Information Science.

Knuth, D. E. (1983). *Literate Programming* (Report Number STAN-CS-82–981). Palo Alto, CA: Stanford University, Department of Computer Science.

---

46. A statistical problem arises in benchmark tests with users that does not normally arise with hardware: unlike computer hardware one can neither get identical people (so that single measurements generalize reliably) nor run a test twice on the same people with identical results (because of learning effects). Even when we understand the causes of variation well enough to apply statistics with confidence, this will still mean running large numbers of trials where one would have been sufficient to benchmark a machine.

Lewis, C. (1983, December). *The 'thinking-aloud' method in interface evaluation.* Tutorial Number 4, presented at the CHI '83 Conference on Human Factors in Computing Systems, Boston.

Nievergelt, J. (1982a). Errors in dialog design and how to avoid them. International Zurich Seminar on Digital Communications, IEEE, *Institut fuer Informatik, ETH, 47.*

Nievergelt, J. (1982b). Towards the integrated interactive system: An experiment in man-machine communication. *Institut fuer Informatik, ETH, 47.*

Norman, D. A. (1983). Design principles for human-computer interfaces. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems* (pp. 1-10). New York: ACM.

Perlman, G. (1983). *Software tools for user-interface development.* Presented at the Summer USENIX Conference, Toronto, Canada.

Roach, J., Hartson, H. R., Ehrich, R. W., Yunte, T., & Johnson, D. H. (1982). DMS: A comprehensive system for managing human-computer dialogue. *Proceedings of the CHI '82 Human Factors in Computer System Conference,* (pp. 102-105). Gaithersburg, MD.

Root, R. W., & Draper, S. (1983). Questionnaires as a software evaluation tool. In A. Janda (Ed.), *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems* (pp. 83-87). New York: ACM.

Sheil, B. (1983). Power tools for programmers. *Datamation, 29,* 131-144.

Shneiderman, B. (1980). *Software psychology: Human factors in computer and information systems.* Cambridge, MA: Winthrop.

Shneiderman, B. (1982). The future of interactive systems and the emergence of direct manipulation. *Behavior and Information Technology, 1,* 237-256.

Smith, D. C., Irby, C., Kimball, R., & Verplank, B. (1982, April). Designing the *Star* user interface. *Byte, 7,* 242-282.

Wasserman, A. I., & Shewmake, D. T. (1982). Rapid prototyping of interactive information systems. *Proceedings of the 2nd SIGOFT Symposium - Workshop on Rapid Prototyping,* Columbia, MD.

Cognitive Science ONR Technical Report List

The following is a list of publications by people in the Cognitive Science Lab and the Institute for Cognitive Science. For reprints, write or call:

Institute for Cognitive Science, C-015
University of California, San Diego
La Jolla, CA 92093
(619) 452-6771

8001. Donald R. Gentner, Jonathan Grudin, and Eileen Conway. *Finger Movements in Transcription Typing.* May 1980.

8002. James L. McClelland and David E. Rumelhart. *An Interactive Activation Model of the Effect of Context in Perception: Part I.* May 1980.

8003. David E. Rumelhart and James L. McClelland. *An Interactive Activation Model of the Effect of Context in Perception: Part II.* July 1980.

8004. Donald A. Norman. *Errors in Human Performance.* August 1980.

8005. David E. Rumelhart and Donald A. Norman. *Analogical Processes in Learning.* September 1980.

8006. Donald A. Norman and Tim Shallice. *Attention to Action: Willed and Automatic Control of Behavior.* December 1980.

8101. David E. Rumelhart. *Understanding Understanding.* January 1981.

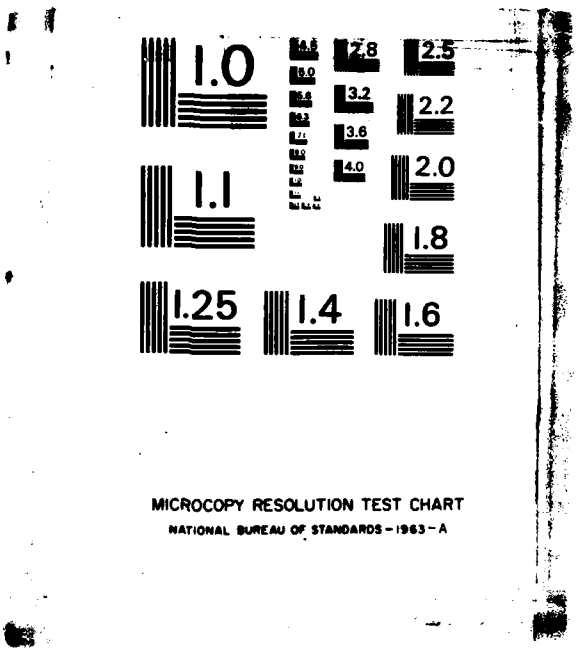
8102. David E. Rumelhart and Donald A. Norman. *Simulating a Skilled Typist: A Study of Skilled Cognitive-Motor Performance.* May 1981.

8103. Donald R. Gentner. *Skilled Finger Movements in Typing.* July 1981.

8104. Michael I. Jordan. *The Timing of Endpoints in Movement.* November 1981.

8105. Gary Perlman. *Two Papers in Cognitive Engineering: The Design of an Interface to a Programming System and MENUNIX: A Menu-Based Interface to UNIX (User Manual).* November 1981.

AD-A141 023 USER CENTERED SYSTEM DESIGN PART 2 COLLECTED PAPERS 2/2
FROM THE UCSD HMI PROJECT(U) CALIFORNIA UNIV SAN DIEGO
LA JOLLA INST FOR COGNITIVE SCIENCE MAR 84 ICS-8402
UNCLASSIFIED N00014-79-C-0323 F/G 5/8 NL



8106. Donald A. Norman and Diane Fisher. *Why Alphabetic Keyboards Are Not Easy to Use: Keyboard Layout Doesn't Much Matter.* November 1981.

8107. Donald R. Gentner. *Evidence Against a Central Control Model of Timing in Typing.* December 1981.

8201. Jonathan T. Grudin and Serge Larochelle. *Digraph Frequency Effects in Skilled Typing.* February 1982.

8202. Jonathan T. Grudin. *Central Control of Timing in Skilled Typing.* February 1982.

8203. Amy Geoffroy and Donald A. Norman. *Ease of Tapping the Fingers in a Sequence Depends on the Mental Encoding.* March 1982.

8204. LNR Research Group. *Studies of Typing from the LNR Research Group: The role of context, differences in skill level, errors, hand movements, and a computer simulation.* May 1982.

8205. Donald A. Norman. *Five Papers on Human-Machine Interaction.* May 1982.

8206. Naomi Miyake. *Constructive Interaction.* June 1982.

8207. Donald R. Gentner. *The Development of Typewriting Skill.* September 1982.

8208. Gary Perlman. *Natural Artificial Languages: Low-Level Processes.* December 1982.

8301. Michael C. Mozer. *Letter Migration in Word Perception.* April 1983.

8302. David E. Rumelhart and Donald A. Norman. *Representation in Memory.* June 1983.

8303. The HMI Project at University of California, San Diego. *User Centered System Design: Part I, Papers for the CHI 1983 Conference on Human Factors in Computer Systems.* November 1983.

8304. Paul Smolensky. *Harmony Theory: A Mathematical Framework for Stochastic Parallel Processing.* December 1983.

8401. Stephen W. Draper and Donald A. Norman. *Software Engineering for User Interfaces.* January 1984.

8402. The UCSD HMI Project. *User Centered System Design: Part II, Collected Papers.* March 1984.

8403. Paul Smolensky and Mary S. Riley. *Harmony Theory: Problem Solving, Parallel Cognitive Models, and Thermal Physics.* April 1984.

ICS Technical Report List

The following is a list of publications by people in the Institute for Cognitive Science. For reprints, write or call:

Institute for Cognitive Science, C-015
University of California, San Diego
La Jolla, CA 92093
(619) 452-6771

8301. David Zipser. *The Representation of Location.* May 1983.

8302. Jeffrey Elman & Jay McClelland. *Speech Perception as a Cognitive Process: The Interactive Activation Model.* April 1983.

8303. Ron Williams. *Unit Activation Rules for Cognitive Networks.* November 1983.

8304. David Zipser. *The Representation of Maps.* November 1983.

8305. The HMI Project. *User Centered System Design: Part I, Papers for the CHI '83 Conference on Human Factors in Computer Systems.* November 1983.

8306. Paul Smolensky. *Harmony Theory: A Mathematical Framework for Stochastic Parallel Processing.* December 1983.

8401. Stephen W. Draper and Donald A. Norman. *Software Engineering for User Interfaces.* January 1984.

8402. The UCSD HMI Project. *User Centered System Design: Part II, Collected Papers.* March 1984.

8403. Steven L. Greenspan. *Reference Comprehension: A Topic-Comment Analysis of Sentence-Picture Verification.* April 1984.

8404. Paul Smolensky and Mary S. Riley. *Harmony Theory: Problem Solving, Parallel Cognitive Models, and Thermal Physics.* April 1984.

Navy

1 Robert Ahlers
Code N711
Human Factors Laboratory
NAVTRASYSCEN
Orlando, FL 32813

1 Dr. Ed Aiken
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Robert Blanchard
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Nick Bond
Office of Naval Research
Liaison Office, Far East
APO San Francisco, CA 96503

1 Dr. Richard Carlano
Navy Research Laboratory
Code 7530
Washington, DC 20375

1 Dr. Fred Chang
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Stanley Collyer
Office of Naval Technology
800 N. Quincy Street
Arlington, VA 22217

1 CDR Mike Curran
Office of Naval Research
800 N. Quincy St.
Code 270
Arlington, VA 22217

1 Dr. Jude Franklin
Code 7510
Navy Research Laboratory
Washington, DC 20375

1 Dr. Jim Hollan
Code 14
Navy Personnel R & D Center
San Diego, CA 92152

1 Dr. Ed Hutchins
Navy Personnel R&D Center
San Diego, CA 92152

Navy

1 Dr. Norman J. Kerr
Chief of Naval Technical Training
Naval Air Station Memphis (75)
Millington, TN 38054

1 Dr. Peter Kincaid
Training Analysis & Evaluation Group
Dept. of the Navy
Orlando, FL 32813

1 Dr. William L. Maloy (02)
Chief of Naval Education and Training
Naval Air Station
Pensacola, FL 32508

1 Dr. Jim McLachlan
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. James McMichael
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr William Montague
NPRDC Code 13
San Diego, CA 92152

1 Library, Code P201L
Navy Personnel R&D Center
San Diego, CA 92152

1 Technical Director
Navy Personnel R&D Center
San Diego, CA 92152

6 Commanding Officer
Naval Research Laboratory
Code 2627
Washington, DC 20390

1 Office of Naval Research
Code 433
800 N. Quincy Street
Arlington, VA 22217

6 Personnel & Training Research Group
Code 442PT
Office of Naval Research
Arlington, VA 22217

Navy

1 Dr. Gary Poock
Operations Research Department
Code 55PK
Naval Postgraduate School
Monterey, CA 93940

1 Dr. Robert G. Smith
Office of Chief of Naval Operations
OP-987H
Washington, DC 20350

1 Dr. Alfred F. Smode, Director
Department D-7
Naval Training Equipment Center
Orlando, FL 32813

1 Dr. Richard Snow
Liaison Scientist
Office of Naval Research
Branch Office, London
Box 39
FPO New York, NY 09510

1 Dr. Richard Sorensen
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Frederick Steinheiser
CNO - OP115
Navy Annex
Arlington, VA 20370

1 Dr. Thomas Sticht
Navy Personnel R&D Center
San Diego, CA 92152

1 Roger Weissinger-Baylon
Department of Administrative Sciences
Naval Postgraduate School
Monterey, CA 93940

1 Dr John H. Wolfe
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Wallace Wulfeck, III
Navy Personnel R&D Center
San Diego, CA 92152

Marine Corps

1 H. William Greenup
Education Advisor (E031)
Education Center, MCDEC
Quantico, VA 22134

1 Special Assistant for Marine
Corps Matters
Code 100M
Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217

1 DR. A.L. SLAFKOSKY
SCIENTIFIC ADVISOR (CODE RD-1)
HQ, U.S. MARINE CORPS
WASHINGTON, DC 20380

ONR Distribution List

Army

1 Technical Director
U. S. Army Research Institute for the Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Beatrice J. Farr
U. S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Harold F. O'Neil, Jr.
Director, Training Research Lab
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Commander, U.S. Army Research Institute for the Behavioral & Social Sciences
ATTN: PERI-OR (Dr. Judith Orasanu)
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Joseph Psotka, Ph.D.
ATTN: PERI-IC
Army Research Institute
5001 Eisenhower Ave.
Alexandria, VA 22333

1 Dr. Robert Sasmor
U. S. Army Research Institute for the Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

Air Force

1 U.S. Air Force Office of Scientific Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, DC 20332

1 Bryan Dallman
AFHRL/LRT
Lowry AFB, CO 80230

1 Dr. Alfred R. Fregly
AFOSR/NL
Bolling AFB, DC 20332

1 Dr. Genevieve Haddad
Program Manager
Life Sciences Directorate
AFOSR
Bolling AFB, DC 20332

1 Dr. John Tangney
AFOSR/NL
Bolling AFB, DC 20332

1 Dr. Joseph Yasutake
AFHRL/LRT
Lowry AFB, CO 80230

Department of Defense

12 Defense Technical Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC

1 Military Assistant for Training and Personnel Technology
Office of the Under Secretary of Defense for Research & Engineering
Room 3D129, The Pentagon
Washington, DC 20301

1 Major Jack Thorpe
DARPA
1400 Wilson Blvd.
Arlington, VA 22209

1 Dr. Robert A. Weber
OUSDRE (ELS)
The Pentagon, Room 3D129
Washington, DC 20301

Civilian Agencies

1 Dr. Patricia A. Butler
NIE-BRN Bldg, Stop 6 7
1200 19th St., NW
Washington, DC 20208

1 Dr. Andrew R. Molnar
Office of Scientific and Engineering Personnel and Education
National Science Foundation
Washington, DC 20550

1 Dr. Everett Palmer
Mail Stop 239-3
NASA-Ames Research Center
Moffett Field, CA 94035

1 Dr. Mary Staddard
C 10, Mail Stop 8296
Los Alamos National Laboratories
Los Alamos, NM 87545

1 Dr. Edward C. Weiss
National Science Foundation
1800 G Street, NW
Washington, DC 20550

1 Dr. Joseph L. Young, Director
Memory & Cognitive Processes
National Science Foundation
Washington, DC 20550

Private Sector

1 Dr. John R. Anderson
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. Alan Baddeley
Medical Research Council
Applied Psychology Unit
15 Chaucer Road
Cambridge CB2 2EF
ENGLAND

1 Eva L. Baker
Director
UCLA Center for the Study of Evaluation
145 Moore Hall
University of California, Los Angeles
Los Angeles, CA 90024

1 Dr. Avron Barr
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Dr. Menucha Birenbaum
School of Education
Tel Aviv University
Tel Aviv, Ramat Aviv 69978
Israel

1 Dr. John Black
Yale University
Box 11A, Yale Station
New Haven, CT 06520

1 Dr. John S. Brown
XEROX Palo Alto Research Center
3333 Coyote Road
Palo Alto, CA 94304

1 Dr. Glenn Bryan
6208 Poe Road
Bethesda, MD 20817

1 Dr. Bruce Buchanan
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Dr. Jaime Carbonell
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

Private Sector

1 Dr. Pat Carpenter
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. Davida Charney
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

1 Dr. Michelene Chi
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. William Clancey
Department of Computer Science
Stanford University
Stanford, CA 94306

1 Dr. Michael Cole
University of California
at San Diego
Laboratory of Comparative
Human Cognition - D003A
La Jolla, CA 92093

1 Dr. Allan M. Collins
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

1 Dr. Emmanuel Donchin
Department of Psychology
University of Illinois
Champaign, IL 61820

1 Dr. Thomas H. Duffy
Department of English
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

1 ERIC Facility-Acquisitions
4833 Rugby Avenue
Bethesda, MD 20014

1 Dr. Anders Ericsson
Department of Psychology
University of Colorado
Boulder, CO 80309

Private Sector

1 Dr. Wallace Feurzeig
Department of Educational Technology
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02238

1 Dr. Gerhard Fischer
University of Oregon
Department of Computer Science
Eugene, OR 97403

1 Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

1 Dr. Michael Genesereth
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Dr. Dan Gentner
Center for Human Information Processing
University of California, San Diego
La Jolla, CA 92093

1 Dr. Dedre Gentner
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02138

1 Dr. Robert Glaser
Learning Research & Development Center
University of Pittsburgh
3939 O'Hara Street
PITTSBURGH, PA 15260

1 Dr. Joseph Goguen
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

1 Dr. Daniel Gopher
Faculty of Industrial Engineering
& Management
TECHNION
Haifa 32000
ISRAEL

1 DR. JAMES G. GREENO
LRDC
UNIVERSITY OF PITTSBURGH
3939 O'HARA STREET
PITTSBURGH, PA 15213

Private Sector

1 Dr. Barbara Hayes-Roth
Department of Computer Science
Stanford University
Stanford, CA 95305

1 Dr. Frederick Hayes-Roth
Teknowledge
525 University Ave.
Palo Alto, CA 94301

1 Dr. Joan I. Heller
Graduate Group in Science and
Mathematics Education
c/o School of Education
University of California
Berkeley, CA 94720

1 Dr. James R. Hoffman
Department of Psychology
University of Delaware
Newark, DE 19711

1 American Institutes for Research
1055 Thomas Jefferson St., N.W.
Washington, DC 20007

1 Dr. Kristina Hooper
Corporate Research, ATARI
1196 Borregas
Sunnyvale, CA 94086

1 Glenda Greenwald, Ed.
Human Intelligence Newsletter
P. O. Box 1163
Birmingham, MI 48012

1 Dr. Earl Hunt
Dept. of Psychology
University of Washington
Seattle, WA 98105

1 Robin Jeffries
Computer Research Center
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

1 Dr. Marcel Just
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

ONR Distribution List

Private Sector

1 Dr. David Klahr
Department of Psychology
University of Arizona
Tucson, AZ 85721

1 Dr. Stephen Kosslyn
1236 William James Hall
33 Kirkland St.
Cambridge, MA 02138

1 Dr. Pat Langley
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. Jill Larkin
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Dr. Alan Leagold
Learning R&D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

1 Dr. Jim Levin
University of California
at San Diego
Laboratory for Comparative
Human Cognition - D003A
La Jolla, CA 92093

Dr. Marcia C. Linn
Lawrence Hall of Science
University of California
Berkeley, CA 94720

1 Dr. Ann Lyon
P. O. Box 44
Higley , AZ 85236

1 Dr. Jay McClelland
Department of Psychology
MIT
Cambridge, MA 02139

1 Dr. James R. Miller
Computer*Thought Corporation
1721 West Plano Highway
Plano, TX 75075

Private Sector

1 Dr. Kurt Miller
Computer*Thought Corporation
1721 West Plano Parkway
Plano, TX 75075

1 Dr. Tom Moran
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

1 Dr. Allen Munro
Behavioral Technology Laboratories
1845 Elena Ave., Fourth Floor
Redondo Beach, CA 90277

1 Dr. Jesse Orlansky
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311

1 Dr. Nancy Pennington
University of Chicago
Graduate School of Business
1101 E. 58th St.
Chicago, IL 60637

1 DR. PETER POLSON
DEPT. OF PSYCHOLOGY
UNIVERSITY OF COLORADO
BOULDER, CO 80309

1 Dr. Lynn Reder
Department of Psychology
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

1 Dr. Fred Reif
Physics Department
University of California
Berkeley, CA 94720

1 Dr. Lauren Resnick
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 1521

1 Dr. Jeff Richardson
Denver Research Institute
University of Denver
Denver, CO 80208

Private Sector

1 Gary S. Riley
Program in Cognitive Science
Center for Human Information Processing
University of California, San Diego
La Jolla, CA 92093

1 Dr. Andrew M. Rose
American Institutes for Research
1055 Thomas Jefferson St. NW
Washington, DC 20007

1 Dr. Ernst Z. Rothkopf
Bell Laboratories
Murray Hill, NJ 07974

1 Dr. William B. Rouse
Georgia Institute of Technology
School of Industrial & Systems
Engineering
Atlanta, GA 30332

1 Dr. David Rumelhart
Center for Human Information Processing
Univ. of California, San Diego
La Jolla, CA 92093

1 Dr. Michael J. Saxon
Perceptronics, Inc
6271 Variel Avenue
Woodland Hills, CA 91364

1 Dr. Roger Schank
Yale University
Department of Computer Science
P.O. Box 2158
New Haven, CT 06520

1 Dr. Walter Schneider
Psychology Department
603 E. Daniel
Champaign, IL 61820

1 Dr. Alan Schoenfeld
Mathematics and Education
The University of Rochester
Rochester, NY 14627

1 Dr. Colin Sheppard
Applied Psychology Unit
Admiralty Marine Technology Est.
Teddington, Middlesex
United Kingdom

Private Sector

1 Dr. H. Wallace Sinaiko
Program Director
Manpower Research and Advisory Services
Smithsonian Institution
801 North Pitt Street
Alexandria, VA 22314

1 Dr. Edward E. Smith
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

1 Dr. Elliott Soloway
Yale University
Department of Computer Science
P.O. Box 2158
New Haven, CT 06520

1 Dr. Kathryn T. Spoehr
Psychology Department
Brown University
Providence, RI 02912

1 Dr. Robert Sternberg
Dept. of Psychology
Yale University
Box 11A, Yale Station
New Haven, CT 06520

1 Dr. Albert Stevens
Bolt Beranek & Newman, Inc.
10 Moulton St.
Cambridge, MA 02238

1 Dr. David Stone
KAJ Software, Inc.
3420 East Shea Blvd.
Suite 161
Phoenix, AZ 85028

1 Dr. Perry W. Thorndyke
Perceptronics, Inc.
545 Middlefield Road, Suite 140
Menlo Park, CA 94025

1 Dr. Douglas Towne
Univ. of So. California
Behavioral Technology Labs
1845 S. Elena Ave.
Redondo Beach, CA 90277

Private Sector

1 Dr. Kurt Van Lehn
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

1 Dr. Keith T. Wescourt
Perceptronics, Inc.
545 Middlefield Road, Suite 140
Menlo Park, CA 94025

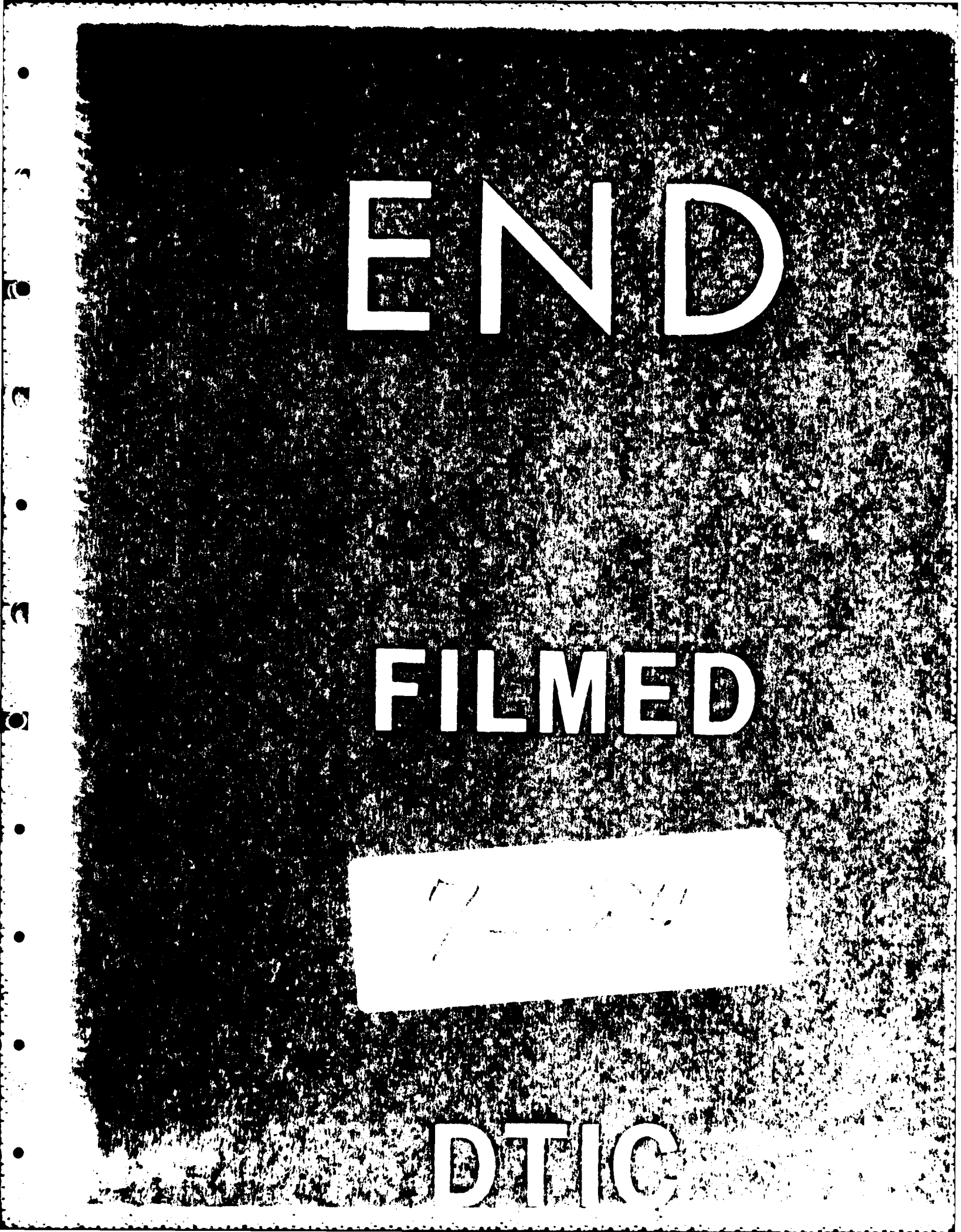
1 William B. Whitten
Bell Laboratories
2D-440
Holmdel, NJ 07733

1 Dr. Thomas Wickens
Department of Psychology
Franz Hall
University of California
405 Hilgards Avenue
Los Angeles, CA 90024

1 Dr. Mike Williams
Intellimetrics
124 University Avenue
Palo Alto, CA 94301

1 Dr. Joseph Wohl
Alphatech, Inc.
2 Burlington Executive Center
111 Middlesex Turnpike
Burlington, MA 01803

ONR Distribution List


END
FILMED
DTIC